

AD-A139 983

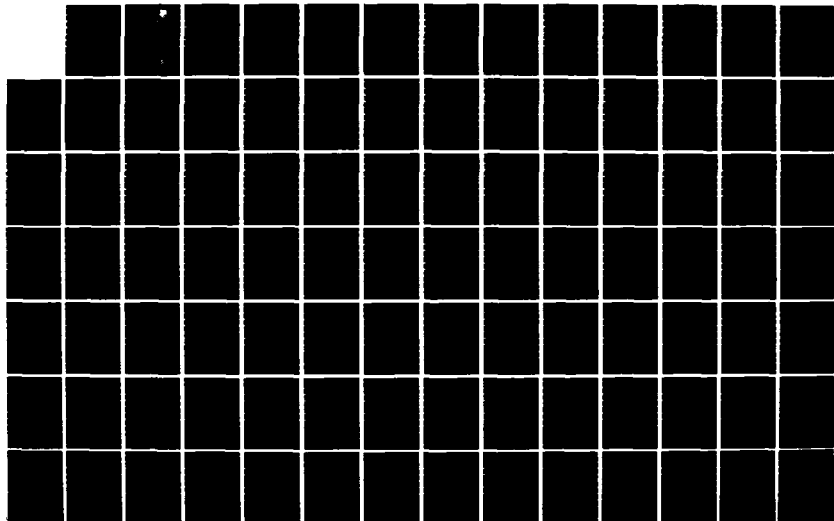
CRONUS A DISTRIBUTED OPERATING SYSTEM(U) BOLT BERANEK
AND NEWMAN INC CAMBRIDGE MA R SCHANTZ ET AL. DEC 83
BBN-5261 RADC-TR-83-255 F30602-81-C-0132

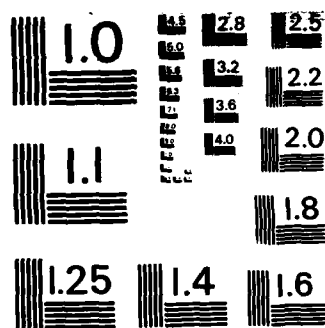
1/3

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-83-255

Interim Technical Report #2

December 1983



CRONUS, A DISTRIBUTED OPERATING SYSTEM

AD A139983

Bolt Beranek and Newman, Inc.

**R. Schantz, B. Woznick, G. Bono, E. Burke, S. Geyer, M. Hoffman,
W. MacGregor, R. Sands, R. Thomas and S. Toner**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC FILE COPY

**ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441**


**DTIC
ELECTE
S APR 11 1984 D**

84 04 11 019

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-255 has been reviewed and is approved for publication.

APPROVED:



THOMAS F. LAWRENCE
Project Engineer

APPROVED:



JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:



JOHN A. RITZ
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-255	2. GOVT ACCESSION NO. AD-A137983	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CRONUS, A DISTRIBUTED OPERATING SYSTEM	5. TYPE OF REPORT & PERIOD COVERED Interim Technical Rpt No. 2 82 July - 82 December	
	6. PERFORMING ORG. REPORT NUMBER BBN Report No. 5261	
7. AUTHOR(s) R. Schantz E. Burke W. MacGregor S. Toner B. Woznick S. Geyer R. Sands G. Bono M. Hoffman R. Thomas	8. CONTRACT OR GRANT NUMBER(s) F30602-81-C-0132	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman, Inc. 10 Moulton Street Cambridge MA 02238	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 25300107	
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COTD) Griffiss AFB NY 13441	12. REPORT DATE December 1983	
	13. NUMBER OF PAGES 266	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release, distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engi. : Thomas F. Lawrence (COTD)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Cronus Distributed Operating System Object Mode Local Network		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report contains the preliminary system/subsystem specification for the Cronus Distributed Operating System. It also reports progress in the development of the underlying support system components, the measurement of local network performance, and the structuring of system messages.		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Table of Contents

1	Report Overview.....	1
2	Project Overview.....	1
3	Summary of Recent Project Activity.....	2

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist.	Avail and/or Special
A/1	



Executive Summary

1 Report Overview

This is the second interim technical report for contract F30602-81-C-0132, entitled "DOS Design and Implementation." The system being developed under this effort has been given the name Cronus. This report discusses project activities during the period of July 1982 to December 1982.

This report is divided into two portions:

Part A: This part is the current version of the Cronus Advanced Development Model (ADM) System/Subsystem Design. A draft of this document was prepared for the Preliminary Design Review, which was held in Cambridge on November 15-16, 1982 in Cambridge MA. The version included in Part B below has been extensively revised, and reflects modifications in the many details of the system that have been made since the draft was prepared. This part is available separately as BBN Report 5260.

Part B: This part consists of a series of short notes and reports of activities performed during the period. Principal among these are discussions of the various activities supporting the development of the system, and of the progress on the components of the system support environment: gce, network, C70 constituent operating system modifications.

2 Project Overview

The object of this project is to define, design, implement and test an Advanced Development Model for a distributed operating system. The DOS controls the interactions among collections of computers interconnected via high-speed local area network technology. The overall function of the DOS is to integrate the various data processing subsystems into a coherent, responsive and reliable system. The system is to include the following functions: system monitoring, reliability and survivability, access control and authentication, and a uniform command language. In addition, the system is to provide support for the following system services: uniform file system,

electronic mail message distribution, data translation, and interactive access to remote programs.

The project activity can be subdivided into five major categories:

1. Definition of the distributed operating system concept and its function as they apply to this effort.
2. Selection of predominantly off-the-shelf hardware and software components to represent the foundation of a demonstration DOS system.
3. Design of the DOS conceptual structure and its functional elements.
4. Implementation of the design, culminating with the integration of implementation units into a complete Advanced Development Model for a distributed operating system.
5. Evaluation of the concepts and realization of the DOs in the environment of the ADM by means of test procedures and practical demonstrations.

The results of the definition and selection phases of the project have previously been reported in Cronus, A Distributed Operating System: Interim Technical Report No. 1, BBN Report No. 5086.

3 Summary of Recent Project Activity

Some of our major accomplishments during the preceding period include the following:

- o completed design of Cronus System Structure and first phase design for all major system components
- o prepared System/Subsystem report
- o completed the integration of the Ethernet local area network into the GCE, the VAX/VMS and the C/70 UNIX hosts
- o completed the integration of IP and TCP protocols into the GCE, VAX/VMS and the C/70 UNIX and interfaced this software to the Ethernet layers using the Virtual Local Network concepts

- o completed a CMOS-based Telnet program to support interactive access to other cluster hosts from the GCE
- o completed the integration of a disk subsystem into the GCE CMOS System
- o completed the design and part of the implementation for a elementary file system for the GCE, which is to serve as the base implementation for the Cronus file system.
- o completed a set of performance tests to evaluate the Ethernet hardware and software, as well as IP and TCP implementations
- o developed and installed a system configuration management plan for source code and documentation
- o developed code for and assembled library functions needed to support the development of Cronus system components
- o established standards and approaches to achieve the high degree of program portability required by our system implementation approach

Report 5261

PART A

Cronus, A Distributed Operating System:
Preliminary SYSTEM/SUBSYSTEM Specification

M. Hoffman, W. MacGregor, R. Schantz,
R. Thomas, E. Burke and B. Woznick

Prepared for:

Rome Air Development Center
Griffiss Air Force Base

Table of Contents

	A-
1 Introduction.....	1
2 Cronus Project Overview.....	3
2.1 Project Objectives.....	3
2.2 Points of Emphasis.....	3
2.3 System Phases.....	4
2.4 The Cronus Hardware Architecture.....	5
2.4.1 System Environment.....	5
2.4.2 Host Classes.....	6
2.4.3 System Access.....	6
2.4.4 Local Area Network.....	7
2.4.5 Types of Hosts.....	8
2.4.6 Cronus Clusters and the Internet.....	9
2.4.7 The Advanced Development Model.....	9
3 System Overview.....	11
3.1 System Concept.....	11
3.2 The Cronus Object Model.....	12
3.3 System Objects.....	15
3.4 The Cronus File System.....	16
3.5 Cronus Process Management.....	17
3.6 Device Integration.....	18
3.7 Cronus Symbolic Catalog.....	18
3.8 User Identities and Access Control.....	19
3.9 Important Subsystems.....	19
3.10 The Layering of Protocols in Cronus.....	20
4 Object Management.....	22
4.1 General Object Model.....	22
4.2 Object Identification.....	25
4.2.1 Cronus Name Spaces.....	25
4.2.2 Accessing Objects.....	29
4.2.3 Summary of the Cronus UID Name Space.....	30
4.3 Operations On Objects.....	31
4.3.1 Primitive Operations and Objects.....	31
4.3.2 Message Communication Support.....	33
4.4 Object System Implementation.....	34
4.4.1 The Operation Switch.....	34
4.4.2 The Operation Switch Interfaces.....	38
4.4.3 The Implementation of SendToHost and Receive.....	39
4.4.4 The General Invocation Sequence.....	42
4.4.5 The Use of UID Location.....	44
5 Process Management.....	45
5.1 Cronus Processes.....	45

5.1.1	Introduction.....	45
5.1.2	Cronus Process Types -- Overview.....	46
5.1.3	The Operations on Objects of Type CT_Primal_Process	
	47
5.1.4	Operations on Objects of type CT_Host.....	50
5.2	Program Carrier.....	52
5.2.1	Objects of Type CT_Program_Carrier.....	52
5.2.2	Operations on Objects of Type CT_Program_Carrier	
	52
5.2.3	The Program Carrier Manager Operations.....	55
5.2.4	Bindings Between Processes.....	56
6	Interprocess Communication.....	58
6.1	Overview.....	58
6.2	Message Structure.....	58
6.2.1	Objectives.....	58
6.2.2	Message Structure Conventions.....	59
6.2.2.1	Self-Description.....	60
6.2.2.2	Language Integration.....	60
6.2.2.3	Data Type Support.....	61
6.2.2.4	Performance.....	62
6.2.3	The Cronus Message Structure Facility.....	63
6.2.4	The Standard External Representation.....	63
6.2.5	Canonical Types.....	65
6.3	Higher Levels of Interprocess Communication.....	68
6.3.1	Message Patterns.....	69
6.3.2	Stream IPC.....	72
7	Authentication, Access Control, and Security.....	74
7.1	Introduction.....	74
7.2	The Cronus Access Control Concept.....	75
7.2.1	Decomposition of the Access Control Problem	
	75
7.2.2	Authorization.....	77
7.2.3	Identification in Cronus.....	78
7.3	Authentication Manager.....	81
7.4	Objects Related to Authorization.....	81
7.5	Operations on Authorization Related Objects.....	83
7.5.1	Operations on the Object of type CT_Authentication_Data	
	83
7.5.2	Operations on Objects of type CT_Principal.....	84
7.5.3	Operations on Objects of type CT_Group.....	85
7.5.4	Operations on Objects of Other Types.....	86
7.5.5	Operation of the Access Control Authorization Function	
	86

7.6	Host Registration.....	88
8	Cronus Primal File System.....	90
8.1	Cronus Primal Files.....	90
8.1.1	Executable Files.....	94
8.2	Crash Recovery Properties.....	95
8.3	Operations for Objects of type CT_Primal_File	
	95
8.3.1	Operations on Object of Type CT- Primal_File_System	
	96
9	Symbolic Naming.....	99
9.1	The Cronus Symbolic Name Space.....	99
9.1.1	General Syntactic Conventions.....	99
9.1.2	Types of Objects Cataloged.....	100
9.1.3	Directories and Links.....	101
9.2	Objects Related to the Catalog.....	104
9.2.1	Objects of Type CT_Catalog_Entry.....	104
9.2.2	Objects of Type CT_Directory.....	106
9.2.3	Objects of Type CT_Symbolic_Link.....	106
9.2.4	Objects of Type CT_External_Linkage.....	106
9.3	Catalog Operations.....	107
9.3.1	Objects of Type CT_Catalog_Entry.....	107
9.3.2	Objects of Type CT_Directory.....	109
9.3.3	Objects of Type CT_Symbolic_Link.....	110
9.3.4	Objects of Type CT_External_Linkage.....	110
9.3.5	Access Control for Catalog Operations.....	110
9.4	Catalog Implementation.....	111
9.4.1	Introduction.....	112
9.4.2	Implementation of the Catalog Hierarchy.....	112
9.4.3	Distribution of the Catalog.....	113
9.4.3.1	Principles Affecting Distribution.....	113
9.4.3.2	Dispersal Of The Catalog.....	114
9.4.3.3	Dispersal of the Cataloged Object Table.....	115
9.4.3.4	Replication of Catalog Information.....	118
9.4.4	Cronus Catalog Managers.....	120
10	Input/Output.....	123
11	User Interface.....	126
12	Monitoring and Control.....	129
12.1	System Capabilities.....	129
12.2	System Model for Monitoring and Control.....	129
12.3	Structure of the MCS.....	131
12.4	Host Probes, Service Probes, and Network Monitoring	
	132
12.5	Loading and Debugging Support.....	134
12.6	Cronus Initialization.....	134
12.7	Siting the Monitoring and Control System.....	136

12.8	Phased Implementation.....	136
13	Scenarios of Operation.....	137
13.1	Basic User Commands and Functions.....	137
13.2	Registering a New User.....	138
13.3	Login.....	138
13.4	Accessing a File.....	140
13.5	Creating a File.....	141
13.6	Deleting a File.....	143
13.7	Listing a Symbolic Catalog Directory.....	143
13.8	Running a Program.....	144
13.9	Starting a Cronus Service.....	146
14	Cronus Primal System Support.....	148
14.1	Primal System Hardware.....	148
14.2	Virtual Local Network.....	151
14.2.1	Purpose and Scope.....	151
14.2.2	The VLN-to-Client Interface.....	152
14.2.3	A VLN Implementation Based on Ethernet.....	157
14.2.4	VLN Operations.....	162
14.3	Generic Computing Element Operating System.....	164
14.4	Cronus Utilities.....	165
14.4.1	General.....	165
14.4.2	Elementary File System.....	166
14.4.2.1	Introduction.....	166
14.4.2.2	File Formats.....	167
14.4.2.3	Disk Salvaging.....	172
14.4.2.4	EFS File System Operations.....	172
14.4.3	UNO Generation.....	175
14.5	Process Support Library.....	179

FIGURES

	A-
Object System Components.....	35
Operation Switch Interfaces.....	38
The SendToHost-Receive Sequence.....	40
The General Invocation Sequence.....	42
The SER Data Structure.....	64
Specifiers for Keys and Values.....	66
A Two Process Invocation (pseudo-code).....	70
A Multiple Process Invocation (pseudo-code).....	71
Retrieving Access Control Data.....	80
Catalog Hierarchy.....	102
Implementation of Cronus Catalog.....	103
Dispersal of the Catalog.....	116
Secondary Symbolic Access Path.....	121
Structure of the MCS.....	130
Cronus Protocol Layering.....	152
A Virtual Local Network Cluster.....	153
EFS File Table.....	169
EFS File Types.....	170

TABLES

	A-
Cronus Objects.....	23
Access State Compatibility.....	93
Access Rights Required for Catalog Operations.....	111
Software Development Hosts.....	149
Generic Computing Elements -- Typical Configurations	
.....	150
Gateway Configuration.....	151
Internet Address Formats.....	155
VLN Local Address Modes.....	156
An Encapsulated Internet Datagram.....	159

1 Introduction

This report presents the preliminary design for Cronus, the system being developed under the Distributed Operating System Design and Implementation project sponsored by Rome Air Development Center(1). It is intended as an overview of the system structure and as a synopsis of the current system/subsystem decomposition and specification.

A previous report, "Cronus, A Distributed Operating System: Functional Definition and System Concept", BBN Report No. 5041 is intended as a companion to the current report, and the reader is assumed to be familiar with its contents. In Section 2, we briefly review a few of the areas covered in the Functional Definition, and extend them to cover current development plans.

Section 3 presents an overview of the Cronus operating system, stressing the common framework into which its components will fit and the functional decomposition of the system.

Sections 4 through 12 present the design for the various system functions. In a number of areas the design is only partially complete. These sections will form the basis of a continuing and evolving subsystem specification for the various components, throughout the life of the project. Section 13 sketches how the system supports some common functions.

Section 14 is a description of the system environment, including hardware, Virtual Local Network, GCE software, and system utilities and libraries.

(1). This work is being performed under RADC contract No. F30602-81-C-0132

2 Cronus Project Overview

2.1 Project Objectives

The objective of the Cronus project is to build an operating system to organize and control a distributed system architecture. The architecture was partially specified by the statement of work, and further defined during early stages of the project. It is described in the Cronus Functional Description [BBN 5041], and is summarized in Section 2.4. In addition to establishing a system architecture, there are five other major aspects of the Cronus project activities:

1. Select off-the-shelf hardware and software components to create an Advanced Development Model (ADM) prototype configuration for the distributed operating system.
2. Define a model for the system operation, develop the functions of the system, and decompose it into implementation units.
3. Develop the implementation units.
4. Integrate the implementation units into a coherent system, both by adjustments to the functional definitions and by any optimizations necessary to achieve acceptable performance.
5. Evaluate the concepts and realization of the DOS in the Advanced Development Model.

2.2 Points of Emphasis

The Cronus design introduces a coherence and uniformity to a set of otherwise independent and disjoint computer systems. This grouping of machines, operating under the control of a distributed operating system, is called a Cronus cluster. The aim is to provide features comparable to those found in a single, modern centralized operating system for the cluster configuration as a whole. There are two ways of viewing this uniformity and coherence; each plays a role in the Cronus design.

From an end user's point of view, the Cronus DOS provides a single account with access to all integrated system services, a uniform distributed filing system and a uniform program execution facility, which is independent of the site of the activity. From

a programmer's point of view, Cronus provides a uniform interface and access path to the distributed system resources, and supports the initiation and control of distributed computations. More importantly, from both an end user's and programmer's perspective, Cronus provides a common system framework for applications. This means that otherwise independent computerized activities can be constructed so that they are more easily made to work together, despite implementations which cross host and processor-type boundaries.

From an operations and administrative perspective Cronus provides a logically centralized facility for monitoring and controlling all of the connected systems. Functions such as account authorization, user priority, and access control can be applied system-wide rather than individually to each host.

In addition to coherence and uniformity, there are a number of other system design goals. These are:

- o Survivability and integrity of Cronus itself;
- o Scalability to accommodate both small and large configurations;
- o Experimentation with resource management strategies that effect global performance;
- o Component substitutability to allow easy use of alternate functionally equivalent hardware; and
- o Convenient operation and maintenance procedures.

2.3 System Phases

System development consists of three phases. The first phase, coincident with the development of the functional definition, included component selection, installation, interconnection and testing. The second phase includes the design and implementation of the basic system that will provide the uniformity and coherency to the collection of machines. It also provides the framework for the in-depth design, implementation, and experimentation in the other areas of interest (e.g. survivability), which are to occur as the third phase. The second phase design is the subject of the remaining sections of this report.

2.4 The Cronus Hardware Architecture

2.4.1 System Environment

This Cronus environment consists of several parts: the local area network which provides the communications substrate for a Cronus cluster, the set of hosts upon which the Cronus system operates, and a mechanism for connecting a Cronus cluster to the Internet environment and to other Cronus clusters.

Cronus enables a variety of constituent computer systems to operate in an integrated manner. Cronus is distinguished from other distributed operating systems by one or more of the following characteristics:

1. Cronus will run on a group of heterogeneous hosts.
2. Cronus hosts will run operating systems which are largely unmodified. Cronus distributed operating system runs as an adjunct rather than a replacement for the hosts' primary operating systems.
3. Hosts will be included in Cronus with varying degrees of system integration. Some support limited subsets of the services defined by the Cronus environment.
4. The interconnection network is designed on a hierarchical model. A Cronus cluster includes a set of hosts connected by a high-speed, low-latency local network. A set of Cronus clusters may be connected over slower long-haul networks.

The Cronus architecture provides a flexible environment for connecting hosts so that facilities available on one host may be conveniently used from other hosts. It provides two alternative host integration schemes. A host may implement the Cronus Interprocess Communication (IPC) mechanism and have efficient communication and operations with the rest of the Cronus hosts; or it may access the other Cronus hosts through an access machine, which is a simpler, less expensive option for connection of a host, but which may be more limited from a flexibility and performance viewpoint.

2.4.2 Host Classes

Cronus hosts can be divided into three groups: mainframe hosts, Generic Computing Elements (GCEs), and workstations.

The collection of mainframe hosts, each of which serves a number of users simultaneously, includes a variety of machines with unrelated architecture. A mainframe host may be tightly integrated into the system, both offering and using Cronus services and fully implementing Cronus interprocess communication. Alternatively, they may be loosely integrated, offering no services, possibly connecting into Cronus through an access machine which provides communication with the rest of Cronus.

The GCE is the workhorse of Cronus. GCEs are small, dedicated-function computers of a single architecture but varying configuration. They provide access machines, file servers, terminal concentrators, and other basic services. Since all GCEs have the same architecture, they provide a replicated resource which, with the appropriate software, enhances the reliability of basic Cronus functions.

Workstations are powerful, dedicated computers which provide substantial computing power and graphics capability at the disposal of a single user. They differ from mainframes in that they support a single user. They differ from terminals in that they offer a significant computational resources.

2.4.3 System Access

There are a variety of user access paths to Cronus. The most typical is a connection by means of a Cronus terminal concentrator. Users may gain access through Telnet protocols from remote points. Cronus also supports access through terminal access mechanisms on its mainframe hosts. These latter two access paths provide the same interface to the user as the terminal concentrator. Access from a workstation will be different than from a terminal, since the workstation defines the user interface. The user has immediate access to the workstation's capabilities.

2.4.4 Local Area Network

The set of hosts is connected by a local area network. The characteristics of the network are crucial to the success of Cronus, since they determine the kinds of communication and operations that are feasible across host components of Cronus.

The selection of an Ethernet for the local area network for the Advanced Development Model has been described in a recent report [BBN 5086]. This choice was motivated by criteria in the project's statement of work:

1. The network should be suitable to support a distributed operating system,
2. The network should be currently available and economical. Since the Advanced Development Model will not be operated in a military environment, certain constraints applicable to a field-deployable version were considerably relaxed.

The Ethernet was chosen for the local area network substrate for the following reasons:

- o The network must be "high-speed". For the ADM, a network must operate at a minimum of 0.5 Megabits per second (Mbits) with low latency, and higher speeds are desirable. The Ethernet operates at 10 Mbits.
- o Network interfaces to all of the computer systems in the DOS ADM should be available. With the exception of the C70, whose Ethernet interface has been constructed under the present contract, this was the case.
- o The local network must provide a datagram-style service.

The Ethernet fulfills all three requirements and we believe is, at the present time, the most cost-effective network technology which does. In addition, the Ethernet provides broadcast and multicast capabilities which, though not absolute requirements, will be usefully exploited in the system.

The raw Ethernet layer will not be used directly. Cronus will use an abstraction of the Ethernet capabilities which is provided by a Virtual Local Net (VLN) software layer, described in Section 14.2. This permits the Cronus Interprocess Communication (IPC) to use DoD standard 32-bit Internet addresses rather than 48-bit Ethernet addresses. It also frees Cronus from a design commitment to the Ethernet. We expect that future

versions of Cronus will need to be built upon a different local network, such as the Flexible Interconnect, which have reliability, communication security, and ruggedization not available in current commercial products. By designing the VLN layer and building Cronus upon it, it should be easy to substitute any local network that provides the basic transport services required by Cronus.

2.4.5 Types of Hosts

GCEs are implemented in the ADM system by Multibus computers with Sun processor board (the current vendor, one of several, is Forward Technology) processors, large (1/2 megabyte) main memories, an Ethernet controller, and additional hardware (disks, RS-232 ports, etc) needed to support specific functions(2). The Multibus computers were chosen because

1. They are relatively inexpensive, permitting low cost incremental system growth.
2. The Multibus standard guarantees the ability to package individual GCEs in different ways with components from a variety of vendors.
3. New processors and devices are expected to evolve for the Multibus over time.

Utility hosts provide the program development and application execution environment for Cronus. In the ADM, this function will be supported by C70 UNIX systems, and, to a lesser extent, by a VAX 11/750(3). UNIX was chosen due to the rich set of development tools already available for it and the ease of developing new tools and applications. The C70 was chosen

(2). One of the functions we would normally install on a GCE is the Cronus Internet Gateway, which will be installed on an DEC LSI-11 computer instead, because the standard Internet Gateway implementation uses the LSI-11.

(3). The use of the VAX 11/750 in this role is a response to a need to utilize the available hardware to distribute our work over available machines. We do not plan a complete utility host environment, and its use as a utility host should be regarded as a matter of local convenience.

because it is one of the least expensive computers which supports a multi-user UNIX, and because of the in-house expertise and support for the hardware base. A VAX running the VMS operating system was chosen to demonstrate the handling of heterogeneous systems.

2.4.6 Cronus Clusters and the Internet

The goal of the Cronus project is development of a local area network-based distributed operating system. The Cronus cluster will operate in the Internet environment as a class B network. Cronus hosts will support the DoD Internet Protocol (IP) for datagram traffic, and, where connections are required, the DoD Transmission Control Protocol (TCP).

A Cronus cluster is expected to use the Internet environment in two ways. First, access will be provided to Cronus from points in the Internet external to the cluster. Second, the Internet will support communication between distinct Cronus clusters.

2.4.7 The Advanced Development Model

The Advanced Development Model (ADM) of Cronus is the first instantiation of the Cronus hardware and software. It is, as its name suggests, the development testbed for Cronus. The ADM different from later models in several respects. First, it will undergo more rapid change as Cronus is developed, software is implemented, altered, and improved.

In an environment this plastic, reliability and availability must suffer. The ADM cannot be as stable as later Cronus systems are expected to be(4).

The ADM is being assembled from off-the-shelf hardware. This reduces the cost of its components, permits the use of state-of-the-art hardware not yet available in ruggedized versions, and enables us to be more flexible in its design. We

(4). Near the end of the contract, we will conduct a period of testing and evaluation during which we will minimize the amount of change in the system so that we can properly evaluate reliability and availability under more typical operation.

are developing a design with the sufficient flexibility to permit later substitution of more suitable hardware for deployable configurations.

3 System Overview

A distributed operating system manages the resources of a collection of connected computers and defines functions and interfaces available to application programs on system hosts. Cronus provides functions and interfaces similar to those found in any modern, interactive operating system (see the Cronus Functional Definition and System Concept Report [BBN 5041]). Cronus functions, however, are not limited in scope to a single host. Both the invocation of a function and its effects may cross host boundaries. The distributed functions which Cronus supports are:

- o generalized object management
- o process and user session management
- o interprocess communication
- o a distributed file system
- o global name management
- o input/output processing
- o authentication and access control
- o system access
- o user interface
- o system monitoring and control.

This report describes those aspects of the Cronus system design which support these functions. In this section, we introduce the Cronus design and briefly discuss the major elements of the system decomposition.

3.1 System Concept

The primary design goal for Cronus is to provide a uniformity and coherence to its system functions throughout the cluster. Host-independent, uniform access to Cronus objects and services forms the cornerstone for resource sharing that crosses host boundaries.

There are two major aspects to the Cronus design: structural and functional. The structural design is concerned with the common framework in which Cronus entities operate. This framework makes Cronus a system rather than simply a collection of functions. The functional design defines the specific services within this system framework, and is the major focus for system decomposition.

The structural design is based on the abstract object model. A distributed system consists of the interaction of concurrently existing active entities called processes. Processes are objects in the system. Processes reside on hosts which are part of the cluster. Some processes, called object managers, play a special role in implementing other objects of the system. Other processes provide services and functions for the clients of the system. Still other processes run user programs. Processes communicate with each other to form larger abstractions and build more complex objects. At the most fundamental level, communication between processes is through messages sent over a local area network connecting the hosts of the cluster. At higher levels, there exist other forms of communication and abstractions in which the communication is implicit rather than explicit. Taken together, there are four interrelated parts to the Cronus system model:

- o A kernel which supports the basic concepts of the object model: processes, communication with objects, object addressing, and the relationship between object types and manager processes. This part of the system includes facilities for locating an object and controlling access to it.
- o A group of basic object types, along with the object managers which implement them. Basic object types include files, processes, devices, and user records.
- o A paradigm for building and accessing new types of objects, which spells out the methods for integrating new object managers into the system on an equal basis with the basic object managers.
- o A user interface and related utility programs (e.g., file copy) to provide convenient access for both people and programs to the system objects and services.

3.2 The Cronus Object Model

Object typing is the foundation of an important methodology for system decomposition. By introducing the type concept at the lowest levels of the design, we are able to decompose parts of Cronus that would otherwise be massed together under the broad heading of "the operating system." This formal decomposition is an important tool in achieving a high degree of host-

configuration flexibility, which is one of the key advantages of a distributed architecture. In addition, it allows us to use function-specific solutions in the design of the various parts of the system.

A fundamental element of Cronus design is the introduction of two system-wide name spaces for referencing objects. One of these name spaces, the unique identifier (UID) space, provides a context-independent method for the accessing objects. Unique identifiers are fixed-length, numeric quantities, intended for use by programs but unsuitable for people to read, remember, and type. A unique identifier also contains the name of object's type and the name of the host that generated it. The host name is useful as a hint for locating certain classes of system objects.

The unique number generator produces UIDs, and is itself an example of a survivable distributed program. The generator must be survivable, because without it new objects cannot be created, and it must be distributed, because UIDs must be unique across all hosts in the cluster, over the lifetime of the cluster.

The operation switch and its associated software interfaces are part of the kernel of the system. The operation switch supports both the location-independent, uniform invocations of operations on objects and location-independent communication between processes, which are themselves objects. Since processes are system objects with defined operations to send and receive messages, the operation switch provides a host-independent interprocess communication (IPC) facility. This facility supports communication for both the system implementation and user application programs. Above these lowest levels of the Cronus system, objects can be accessed without regard to their location. The design of the operation switch is described in Section 4.

In general, three somewhat different classes of objects will be accessed through the operation switch. These are:

1. Non-Migratory Objects

These are the simplest form of object, which are forever bound to the host which created them. These, objects are often referred to as primal, in the sense that there is no simpler form of Cronus object for this role. An example would be a Primal File, which is permanently bound to its storage site. Primal objects have the property that the host hint embedded in the UID is always valid, and can be used to access the object directly.

2. Migratory Objects

These are objects which that may move from host to host as situations and configurations change. An example would be a directory, which might under a system reconfiguration migrate to an alternate site. Despite the possibility of migration, the global unique identifier for an object remains the same throughout its lifetime. A standard-Cronus mechanism that queries the object managers can locate the current site to complete an object access.

3. Structured and Replicated Objects

These are objects which have more internal structure than a single uniquely identified object. For example, a replicated file would as a unit have a single, global unique identifier, but would have a number of primal file as its constituent parts. In the case of replicated objects the unique object identifier would be recognized by manager processes on each of the sites for the more primitive elements. Replicated objects and the managers that support them are a key element in our approach to system survivability. Invocation of operations against replicated objects involves a selection phase in addition to the location phase described for migrating objects. The object access software implements a rudimentary form of automatic resource management by selecting an appropriate instance of the object for the operation invocation. These resource management selections can be overridden by the accessing process, if desired.

Maintaining the integrity of complex objects is the responsibility of the managers for the type. This means that techniques can be tailored to the patterns of access to the object being maintained. The construction of complex objects out of the more primitive objects is one of the key aspects of Cronus system extensibility.

Uniform access control is another part of the Cronus object model. The object managers and controlling access to the objects they maintain through the use of access control lists. The operation switch assists by reliably stamping the UID of the invoking process on each of its requests.

Up to this point, we have described the method of accessing objects starting from a program-oriented unique identifier. The Cronus system includes a global symbolic name space oriented

toward human use. The primary purpose of this name space is to catalog object names in a manner which is convenient for people to use.

To access a symbolically cataloged object, the accessing agent interacts with the Cronus symbolic catalog manager to find the unique identifier for the object. After it obtains the UID, the accessing agent can then invoke operations on the object.

In Cronus, the conventions for communication between cooperating system entities are based on the message structure facility (MSF). The MSF supports messages structured as collections of key-value pairs. Many keys are standardized to support the object model and basic interprocess communication functions. Examples of standardized keys are operation name, transaction identifier, and error code. Other keys are standardized for particular system services and are published with documentation for these functions. There are also conventions that provide simple transaction protocols, and other features to support flexible message handling and processing. The MSF also standardizes the representation of certain values, which allows the common interpretation of these data items across the collection of heterogeneous Cronus hosts. The MSF design is discussed in Section 6.

3.3 System Objects

The object-oriented system model is extensible along two primary dimensions:

- o new object types can be added to support new requirements or functions, and
- o more complex subtypes of objects can be added to extend existing Cronus types.

To provide the initial operating capability, a number of basic system objects and functions must be developed. These parallel the functions outlined in the Cronus functional definition. They include:

- o File objects and file managers which provide a distributed filing system for both system and client storage and retrieval.

- o Process objects and process managers support the Cronus system and user programmable processes. They may be linked together across the cluster, and connected through interprocess communication to form a user session. User programmable process objects represent another important aspect of system extensibility.
- o Device objects and device managers support the integration of I/O devices into Cronus.
- o User identity objects and a permanent user data base support authentication, access control.
- o Directory objects and catalog managers implement the global symbolic name space.

Much of the Cronus design has been decomposed into the subproblems of designing the components which provide these basic system objects.

3.4 The Cronus File System

Cronus supports several file types. The most basic file is a primal file, which is stored entirely within a single host and is bound to that host throughout its lifetime. Other types of Cronus files, are built from primal files. For example a migratory file can have multiple instances replicated across Cronus hosts for increased availability or enhanced responsiveness, consists of several primal files.

Hosts which contribute storage resources to Cronus must support primal files. The collection of all Cronus files constitutes the Cronus distributed file system. This file system provides the major support for Cronus non-volatile storage requirements. It supports an atomic update concept to aid in the construction of object managers.

There is no single table that list all file objects. Rather, each file manager owns all of the data for the file objects it manages. Cooperation among object managers, and the use of protocols based on broadcast requests to locate objects, make possible a client interface in which knowledge of an object UID is sufficient to access the object regardless of its location. Clients can make file placement decisions themselves if they wish. Alternatively, placement decisions can be made

automatically by file access software. File managers support a protocol for direct access to file data as well as higher-level, complete file transfer protocols. The expected mode of access to Cronus files is to transfer the file data in blocks as needed, much like conventional file system access to disk files(5). Copies of Cronus files are made only to satisfy explicit user requests. The design for the Cronus Primal File System can be found in Section 8.

3.5 Cronus Process Management

There is more than one type of process object in Cronus. Primal processes are the simplest process entities. They are constructed from the process abstraction that exists in the constituent host operating system. This simple form of process is used as a building block for the system implementation. Its simplicity minimizes integration costs for new Cronus host types. Primal processes are too inflexible to be used as vehicles for general application programming. For example, they cannot be loaded dynamically with user programs and they lack flexible process control functions. They are tailored to their well defined system roles.

To satisfy the requirements of application programs, primal processes are augmented with a subtype, the program carrier process. This subtype supports a richer process environment. Program carrier processes can be loaded remotely and started in a manner that is uniform across the cluster. In addition, program carriers support, in a host-independent manner, the kind of flexible control and interconnection of related processes found in modern operating systems.

An important principle behind the Cronus process concept is the additive nature of the common Cronus process semantics. Cronus processes have most of the features natural to the host on which they are built. No attempt is made to hide these features. An application builder has the choice of when to use locally-supported features and when to use standardized Cronus features. Clearly, to the extent that applications choose to adopt Cronus process features, they will be better integrated with the other

(5). This is in contrast to a system such as NSW in which a reference to a file always results in a complete file transfer copy.

cluster processing activities. The Cronus process concept is described in Section 5.

3.6 Device Integration

Input/Output devices, such as line printers, tape drives, and other special purpose devices are important elements in a system configuration. The objective is to make these devices available to the entire cluster. Devices are Cronus objects and are integrated through a Cronus device manager which services the particular type of device. The object system support makes device I/O functions available from anywhere in the cluster. In some cases, for example, for a line printer service, more elaborate interfaces can provide a more convenient access path with specialized features, such as spooling. Device integration is discussed in Section 10.

3.7 Cronus Symbolic Catalog

The Cronus Symbolic Catalog maps user-oriented symbolic names into the program-oriented unique identifiers needed to access Cronus objects. The Cronus catalog implements a global, hierarchical, host-independent name space which can be used to catalog any Cronus object. The catalog is distributed; different hosts manage different parts of the name space. The implementation is logically integrated, however, so any catalog manager process can be asked to perform any of the catalog operations. The upper portion of the hierarchy is replicated to support the flexible assignment of parts of the name space to catalog manager hosts. The symbolic catalog introduces and supports additional system objects such as directories and catalog entries. Symbolic naming in Cronus is discussed in Section 9.

3.8 User Identities and Access Control

Users are represented by system objects, known as principals. Associated with each principal is a data base entry which is a record of information about that user's use of the system. This information supports operations such as authentication, and session initialization. The Authentication Manager is responsible for managing the user data base. The Authentication Manager component services the entire cluster.

The Authentication Manager and the unique identifiers for principals play key roles in the uniform Cronus-wide authentication and access control mechanism. The purpose of Cronus access control is to prevent unauthorized access to Cronus objects. This is done uniformly by associating an access control list with each object. Access is then either granted or denied based on the identity of the principal associated with the accessing agent and the contents of the access control list for the object. How these functions are accomplished in the Cronus distributed system environment is discussed in Section 7.

3.9 Important Subsystems

Subsystems are components which use system-provided features to support user services. Two important subsystems are part of the initial system implementation. These are the user interface subsystem and the monitoring and control subsystem.

The user interface is the component with which the user interacts. One kind of user is the programmer building Cronus applications. An important component of the programming interface is a Program Support Library (PSL) which implements a more convenient and powerful way to use the basic functions provided by object managers. These areas have only been briefly addressed so far in our design. Introductory discussion can be found in Sections 11, 13, and 14.

The monitoring and control subsystem (MCS) makes it possible for an operator to monitor and control the entire cluster configuration from a single console. The functions of the MCS include starting or restarting parts of the Cronus configuration, monitoring its facilities and components, and collecting error reports and statistics. The MCS is based on a functional decomposition across the Cronus configuration rather than a site-based decomposition. The monitoring and control design is

described in Section 12.

3.10 The Layering of Protocols in Cronus

The underlying support for the Cronus cluster architecture is a high-speed local area network. The Ethernet standard has been selected for an interhost transport medium within the initial Cronus configuration. However, two project goals suggest that the Cronus implementation not be based directly on the Ethernet. These are:

- o Substitutability

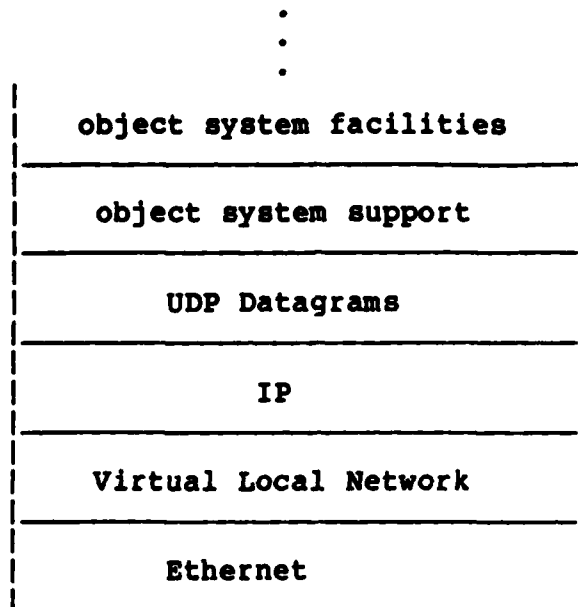
Other instances of a Cronus cluster will not, in all likelihood use the Ethernet as their physical local network. Substitution of another local network should be accomplished with minimum change to system software.

- o Compatibility

The context for this project suggests that it is important to maintain compatibility with the emerging DoD standard Internet protocols.

To accomplish these objectives, we have developed a Virtual Local Network based on Internet Protocol (IP) conventions and a representative set of local area network capabilities. The Virtual Local network is an interhost message transport medium which is independent of the physical local network.

The Virtual Local Network layer is described in section 14.2. It provides a primitive datagram service, compatibility with internet addressing, and independence from the details of the physical local network. VLN datagrams can be specifically addressed, broadcast, or multicast. The VLN guarantees that datagrams are delivered in order (sequenced) when they are delivered at all, and that a datagram is received once or not at all by each intended recipient (non-duplication). The layering of protocols in the system architecture is illustrated in the following diagram.



The IP and UDP layers are the standard internet protocols. UDP provides a datagram service with a 16-bit destination field, and is needed to implement multiplexing of IP datagrams among the Cronus modules below the object system layer. One example of a function below that layer is the unique number generator facility.

4 Object Management

4.1 General Object Model

This section contains an overview of the Cronus object model, from its foundations in UID naming, through the most basic operations on objects. It then presents the design for the operation switch, which provides the underlying support for the object orientation of the system. Processes are objects with defined operations Send and Receive. Therefore, basic system support for low-level message-oriented interprocess communication is part of the operation switch. Section 6 contains the design for higher levels of Cronus interprocess communication.

The object model provides a coherent and uniform framework for the system components of Cronus, and potentially for some application programs which will inhabit a Cronus cluster. A Cronus object has two kinds of features:

- o Required features. Cronus requires certain minimal set of features for each object type.
- o Conventional features. The object model and its associated system components define a number of conventions, which may be adopted by subsystem designers, on a case-by-case basis.

A subsystem designer can depend upon the existence of required features in other system components, and is obligated to provide them in each new component.

A Cronus system design goal is to minimize the number of required features for system entities. This, in turn, minimizes the buy-in costs for new host types. Designating features as conventional rather than required also reduces the potential for conflict between basic Cronus functions and those of Constituent Operating Systems. Conflicts of this type can greatly increase the integration costs for Cronus hosts.

The references [Xerox 1981, Rentsch 1982] discuss the object-oriented model of programming. This section briefly reviews the concepts of inheritance, subtypes, and supertypes and explains their relationship to the Cronus design. The currently defined Cronus Types may be found in Table 1. By convention, Cronus types are designated with a prefix of CT_.

Inheritance, subtypes, and supertypes describe ways that one object type can be derived from another. A new type can be

Object Name	See Section
CT_Cronus_Host	5.1.4
CT_Type_Name	4.2.3
CT_Cronus_Process	5.1.2
CT_Primal Process	5.1.3
CT_Program Carrier	5.2
CT_Cronus_Catalog	9.2
CT_Catalog_Entry	9.2.1
CT_Directory	9.2.2
CT_Symbolic_Link	9.2.3
CT_Eternal Link	9.2.4
CT Cronus File	8.1
CT_Primal File System	8.5
CT_Primal File	8.1
CT_Migratory File	8.1
CT_Dispersed_File	8.1
CT_Executable_File	8.1
CT_Principal	7.5.2
CT_Group	7.5.3
CT_Authentication Data	7.5.1
CT_Session_Data	13.3
CT_Line Printer	10
CT_Elem File System	14
CT_UNIX_Name Space	9.2.4
CT_VMS Name Space	9.2.4

Table 1. Cronus Objects

defined by the way it extends or differs from another object type. The properties that the new type has in common with the old type are said to be inherited; the new type is called a subtype of the old type, and the old type is a supertype of the new one.

More formally, to say that objects of type A inherit the

properties of objects of type B means that the operations which are valid on B objects are also valid on A objects, with the same semantics. If type A objects inherit the properties of type B objects, we say that A is a subtype of B, and B is a supertype of A. The subtype-supertype relationships are transitive; if A is a subtype of B and B is a subtype of C, then A is a subtype of C. For example, CT_Primal File is an object type in the distributed file system; the operations

Open (Primal File UID) reply code

Read (Primal File UID) file data

(...and so on)

act on objects of type CT_Primal File, inspecting or changing the state of a CT_Primal File object. Executable files are primal files, but they also have characteristics not shared by other primal files. The similarities and the differences are captured by defining a type CT_Executable_File as a subtype of CT_Primal File. Objects of type CT_Executable_File inherit the operations valid for all primal files, and also respond to operations unique to executable files, for example, we might have operations such as:

Processor Type (Executable_File UID) returns
(The host class for the executable e.g. VAX)

Resource Requirements (Executable_File UID) returns data

(...and so on)

Subtypes of CT_Executable_File could be defined to distinguish "M68000_Executable" from "VAX_Executable" and other kinds of executable objects with unique properties. For each of these subtypes, the operations defined on objects of the supertypes CT_Primal File and CT_Executable_File would be valid, as well as any new operations defined on the subtype.

Subtype/supertype relationships are statically realized in Cronus, through the cooperation of the object managers and the operation switch. No automatic mechanism is currently provided for inheritance. There are several static implementation techniques that can achieve inheritance. A manager may register several type values with the operation switch, and implement some as subtypes of the others internally. Alternatively, one manager may invoke another through the standard mechanisms.

4.2 Object Identification

4.2.1 Cronus Name Spaces

There are two levels of naming objects within Cronus. Therefore, there are two distinct name spaces, and two levels of cataloguing and name management in Cronus.

At a relatively low level there is a global unique identifier (UID) uniform name space for Cronus objects. Every Cronus object has a UID name. Programs (as opposed to people) are the primary users of this name space. A principal design consideration for the UID name space is to make it easy for programs to use, so UID names are fixed length bit strings.

Although there is no single identifiable catalog supporting the UID name space, the notion of a catalog for UIDs is a useful abstraction. This catalog will be referred as the "UID Table" with the understanding that, in practice, the functions that it supports are implemented by object managers for different object types by means of UID-to-object-descriptor tables which can be thought of as fragments of the UID Table. Every Cronus object is catalogued in a UID table. When a Cronus object is created, an entry is created in a UID table. This entry contains enough information for the manager of the object to access it. Object managers generally support type-dependent operations for creating, manipulating and deleting objects, and for inspection and maintenance of the UID table entries. The Cronus operation switch provides client processes with object-oriented addressing, so merely having the object UID is sufficient to communicate with the object.

At a relatively high level, there is a global symbolic name space for Cronus objects. Symbolic names are more convenient for human users, so the principal design consideration for the Cronus symbolic name space is to make it easy for people to use. Symbolic names are supported by a catalog which will be referred to hereafter as the "Cronus Catalog", which provides a mapping between the symbolic names that people use and the UIDs that are required to actually access the objects. This name space is hierarchically structured as a tree. The tree contains nodes and directed labeled arcs. There is a node called the "root". Each node has exactly one arc pointing to it, and can be reached by traversing exactly one path of arcs from the root node. Nodes in the tree generally represent Cronus objects which have symbolic names. A complete symbolic name is formed by listing the names of the arcs, separated by the punctuation mark ":". For example, :a:b:c is the symbolic name of an object.

Not all Cronus objects have symbolic names, and those that do may have more than one. When an object is given a symbolic name, an entry is made in the Cronus Catalog, and when the name for an object is removed, its entry is removed from the Cronus Catalog. The Cronus Catalog supports Enter, Lookup, and Remove operations. In addition, operations are provided to read and to modify the contents of catalog entries. The catalog entry corresponding to a symbolic name includes the UID of the object named. The Cronus catalog is described in detail in Section 9.

A Cronus unique identifier consists of a pair

<UNO, Type>

where UNO is a 64-bit unique number, and Type is a 16-bit value naming the type of the object. The UNO portion of the UID is uniquely associated with a particular object. Each Cronus service is administratively assigned a unique type and, all types are statically well-known. Since the Type field will encode as many as 65,536 distinct types, there is room for expansion to dynamic types at a later time. By convention, the symbolic names of Cronus types all begin with the prefix "CT_", e.g., CT_Primal File.

The facility which generates unique numbers may be regarded as existing continuously throughout the life of a Cronus configuration, and is accessible to system and application processes. No two requests by client processes for a UNO ever obtain the same UNO, over the entire lifetime of a Cronus cluster. UNOs are guaranteed unique only over the domain of a single cluster.

UNOs are fixed-length strings of length of 64 bits or 8 bytes. They are comparable in size to short symbolic names, and can be easily stored and manipulated on byte-oriented, 16-bit-word, and 32-bit-word machine architectures.

The UNO generation scheme is logically centralized because two generators in the same Cronus cluster must not generate the same UNO. Since we want the facility to be continuously available with high probability to processes distributed over various hosts, the implementation of the UNO generator facility is physically distributed(6)

(6) A description of the design and implementation for the Cronus unique number facility can be found in Section 14.4.

The UNO consists of three fields: a HostNumber, a HostIncarnation and a SequenceNumber. The HostNumber field identifies the machine which generated the UNO. The HostIncarnation is a centrally-generated number (or one which is dependent upon local non-volatile storage and periodic central synchronization) which assures the uniqueness when a host crashes and returns to the system. The SequenceNumber is incremented for each request.

The UNO size, 64 bits, was derived from assumptions about the number of UNOs that could be generated over the lifetime of a Cronus cluster. We assume that the maximum number of hosts in a cluster is 1024, and the maximum lifetime of a DOS cluster is 100 years. The implementation strategy imposes constraints upon the rate at which UNOs can be generated (fewer than 1000 per second per host) and on the rate at which a host can leave and reenter the cluster-wide UNO generation mechanism (about once every 10 seconds).

A Cronus service is implemented from a Cronus process. The UID for the process is a unique identifier of type CT_Primal Process, selected when the process was created. To facilitate communication between accessing agents and Cronus' services, Cronus also assigns a logical name to each service. A logical name is a UID selected from a reserved portion of the UID name space which, is itself designated by a Cronus type, CT_Type_Name. Every Cronus type maps to a logical name UID formed from a 16-bit type by setting the HostNumber field to an arbitrary value, the HostIncarnation field to zero, the SequenceNumber field to the 16-bit type, and the type field to the constant CT_Type_Name.

Cronus provides a pair of functions which can be used to convert between a type name and the logical name for the manager process of that type. These functions are:

NameToType (LogicalNameUID) returns Type

TypeToName (Type) returns LogicalNameUID

Logical names, like types, can be referred to symbolically. By convention, logical names begin with the prefix "CL_". For example, CL_Primal File refers to the process which manages primal files. A logical name can be used to locate or generically address an object manager for its type, or the collection of all managers for the type.

Two basic kinds of UID names for Cronus objects have been

introduced.

1. Specific names.

The specific UID name is assigned when the object is created, by requesting a new UNO. The creating agent may retain the specific name or pass it on to other processes, to allow the possessors of the name to operate on the object.

2. Logical names.

All logical names in Cronus are well-known, and have symbolic equivalents (e.g., CL_Primal File). Because they are statically known, they can be built into the system and application programs and used to establish a rendezvous between processes and standard objects or services.

Specific names are used for objects which can be created and destroyed, and have private state information which is important to the accessor (e.g., a particular file). Logical (generic) names are used to refer to system services (e.g., CL_Primal File refers to the processes which manage Primal Files). System services usually fulfill the same role over long periods of time, and are not created and destroyed by application programs.

Each Cronus service has a unique type and a logical name. In addition, the processes which implement the service have specific object UIDs, since they are process objects. Operations can be invoked using either the logical name for the service or the UID for the manager. For example, the logical name for a Primal File Manager is well-known, and can be used to invoke primal file operations and communicate with the primal file manager independently of the current specific process UID name for the Primal File manager. File operations invoked on the CL Primal File logical name, for example, would be delivered to one or more Primal File managers. In contrast, the process control operations defined on a Primal Process object can only be invoked using the specific UID name of a particular Primal File manager process.

4.2.2 Accessing Objects

Accessing agents interact with object managers using Cronus Interprocess Communication. Access may be initiated in one of two ways:

1. Directly through the UID name space.

The accessing process has the UID of the object, and invokes an operation upon it. The operation switch delivers the requested operation, the UID, and any other parameters to the appropriate object manager. The object manager consults its fragment of the UID Table to access the object as necessary to perform the requested operation.

2. Through the symbolic name space.

The accessing process has a symbolic name for the object. In this case, access is accomplished by consulting the Cronus Catalog to find the UID for the object. Now access to the object can proceed as described in (1) above.

Allowing the symbolic catalog to be by-passed when an object is accessed improves performance and enhances the flexibility of using primitive objects to build complex objects.

Of course, the accessing process must have the UID for the object in order to access it. The cost of achieving these benefits is primarily one of increased implementation complexity:

1. Access control is performed in a decentralized fashion by all of the object managers.
2. Information about objects is distributed among object managers and catalog managers. Care must be taken to ensure that the information about an object is consistent, and if it is not, that the system can operate properly.

4.2.3 Summary of the Cronus UID Name Space

In preparation for the detailed discussion of the operation switch design, we first briefly review the key properties of the Cronus UID name space:

1. UIDs are fixed length bit strings.

A UID is 80 bits long and consists of a 64-bit Cronus Unique Number (UNO) and a 16-bit type specifier.

2. The type of the object named by a UID can be determined from the UID.

The UID has a 16-bit type field. The ability to determine the type of an object solely from its UID is critical to the implementation of the operation switch.

3. The UID for an object is a host independent name for the object.

UIDs are host independent in two ways. First, the UID may be used to refer to the object regardless of the host from which the reference is made. Second, the UID may be used to refer to it regardless of the host (or hosts) which implement the object.

Although it may not happen often in practice, objects may move (or be moved) from host to host. When an object is relocated in this fashion, its UID remains fixed.

4. The UID for an object contains a hint for the location of the object.

The HostNumber field is used as a hint for the host location of the object. This is the host that generated the UNO, and it will frequently be the host responsible for the object. Since some types of objects can move from host to host, the HostNumber field of a UID for such an object does not positively identify its location. When the hint fails for these objects, a Locate operation will find the object. For objects that cannot move (e.g., primal processes, primal files) the hint is guaranteed to be valid.

5. Communication with an object manager can be initiated merely by knowing the type that it manages.

Logical UID names which can be computed from the object type, provide generic addressing.

4.3 Operations On Objects

4.3.1 Primitive Operations and Objects

An operation to be applied to an object is represented as a pair

<OperationName, Parameters>

There are several ways to invoke an operation, but in some sense the most primitive of which is InvokeOnHost. The function supports the invocation of Operation on the object named by ObjectUID on the host with internet address HostAddress.

Invocations do not necessarily cause a reply or acknowledgement to be returned to the invoker. Most operations will follow a request-reply paradigm, but there are important examples of operations that will not. The generation of a reply, and the conventions describing the contents of the reply, are an example of conventional features derived from the request-reply paradigm.

InvokeOnHost can invoke an operation on a host that is remote from the invoking process. The operation switch will attempt to deliver the operation only to the addressed host or hosts. The parameter HostAddress is a Virtual Local Network address (see Section 14.2), and may refer to one host (if it is a VLN specific address), all hosts (if it is the VLN broadcast address), or a subset of hosts (if it is a VLN multicast address). Assume for the present that HostAddress is, in fact, a specific VLN address and refers to just one target host.

The ObjectUID may be a logical name (i.e., a UID of type CT_Type_Name). Logical names can be used to invoke operations on system-defined object managers and service processes, or system-defined objects.

The operation Locate is defined on every object in the system; Locate is a required feature of every object manager. The Locate operation can be invoked in the following way:

InvokeOnHost(HostAddress,ObjectUIDP,

<"Locate",ReplyOption>)

Locate generates a reply from the host HostAddress if the object ObjectUID is present on that host. The ReplyOption parameter may be either Always or PositiveOnly. A value of Always means both positive and negative replies are generated. A value of PositiveOnly means only positive replies are sent back.

If HostAddress is the VLN broadcast address, the Locate operation queries all hosts. Then,

InvokeOnHost(VLNBroadcastAddress,ObjectUIDP,
<"Locate",Always>)

tests all active hosts for ObjectUID, and solicits a reply from all hosts. The invocation

InvokeOnHost(VLNBroadcastAddress,CL_Primal File,
<"Locate",PositiveOnly>)

asks for replies from those hosts supporting the object type CT_Primal File. This operation might be performed in preparation for creating a new primal file.

A library routine will be available which invokes the operation collects the replies and presents them to the process:

Locate(ObjectUIDP,StopWhen,Timeout) returns ReadyList

This routine performs an InvokeOnHost to the VLN broadcast address, on the object named through ObjectUIDP, with the operation <"Locate",PositiveOnly>. The integer StopWhen determines how many replies the routine returns. If StopWhen is a positive integer, the routine will collect replies until StopWhen replies are received, or Timeout seconds have elapsed, whichever comes first. If StopWhen is 0, the routine waits until Timeout seconds have elapsed. In either case, the information contained in the replies is collected and, after editing, a pointer to this list passed back to the caller.

The ReplyList contains the VLN addresses of replying hosts. If the ObjectUIDP refers to a logical name, the reply list will also contain the specific UID of each responding process. Type-specific information may also be present in the ReplyList. The following paragraphs describe a simplified form of this ReplyList, namely, that ReplyList [1] is simply a host address.

The InvokeOnHost and Locate can be combined to define a more

general, host-independent Invoke operation, shown in the pseudocode definition below.

```
operation Invoke(ObjectUID,Operation) is
  Locate(ObjectUID,l,TimeoutConstant)
  if HostList(l) != NULL then
    InvokeOnHost(HostList(l),ObjectUID,Operation)
```

This is the simplest possible form of Invoke; more complex variants, which involve caching UIDs, for example, will be developed. The properties of the Invoke operation will be discussed further below; in particular, there is an important optimization which makes the Locate operation unnecessary for some object types, such as primal files, for which the hint in the UID name is defined to be accurate.

4.3.2 Message Communication Support

In order to describe the design of the operation switch and its role in message-oriented interprocess communication, we must briefly introduce Cronus processes. The Cronus process concept is described in detail in Section 5.

Cronus processes are constructed from constituent host processes (CHPs). The properties of a CHP are defined by the machine architecture and the constituent host operating system (COS). The Cronus process consists of a CHP plus the Cronus process features. The simplest type of Cronus process is the Primal Process (PP). A Primal Process is a CHP with the ability to invoke Cronus operations through InvokeOnHost. In addition, there are operations which can be applied to Primal Process objects, including SendToHost and Receive, which provide a simple message service and basic process control. SendToHost and Receive are implemented by the operation switch.

The SendToHost operation transmits a message from one process to another, and the Receive operation makes a previously sent datagram visible to the recipient.

```
SendToHost(TargetInternetAddress,TargetProcessUID,Text)
```

```
Receive(SourceInternetAddress,SourceProcessUID,Text)
```

are a matching pair of SendToHost and Receive operations. Note that Receive obtains the internet address of the sender, as well as the sender's UID and the message text. The SendToHost command above is considered to be equivalent to

```
InvokeOnHost(TargetInternetAddress,  
             TargetProcessUID,  
             <"SendToHost", Text>)
```

An implementation of Receive employs pseudo-interrupts or other CHP-specific synchronization facilities, not defined here, to build an asynchronous Receive operation.

The more general operation Send is related to SendToHost in the same way that Invoke is related to InvokeOnHost. A simple Send is defined as

```
operation Send(ProcessUID,Text) is  
  Locate(ProcessUID,HostList,1,TimeoutConstant)  
  if HostList(1) /= NULL then  
    SendToHost(HostList(1),ProcessUID,Text)
```

As for Invoke, optimizations are possible.

4.4 Object System Implementation

4.4.1 The Operation Switch

This section describes the framework of the object system implementation on Cronus hosts. Figure 1 illustrates the relevant components on a single host.

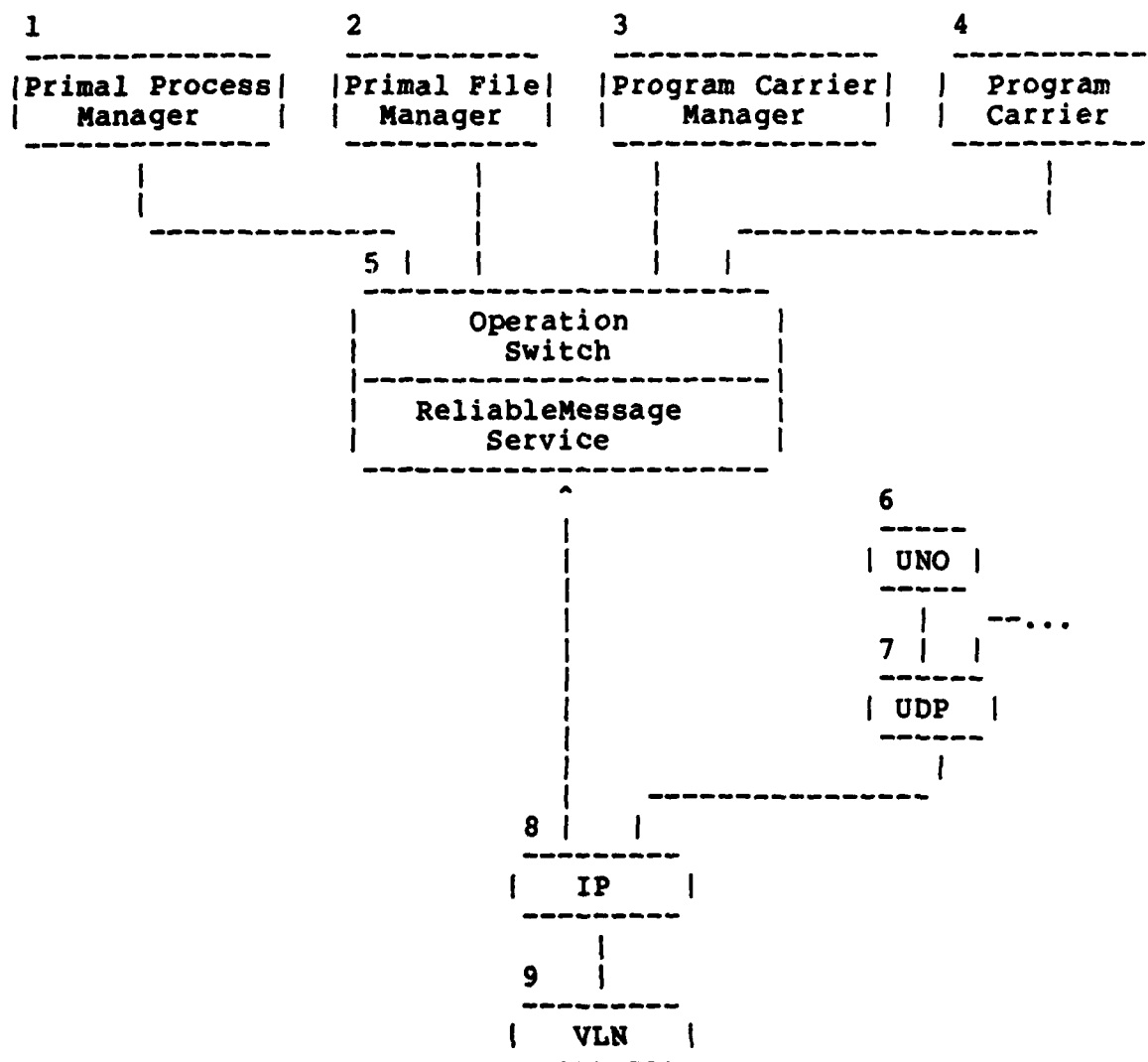


Figure 1 . Object System Components

The boxes in the figure represent abstract modules of the implementation, and do not necessarily map one-to-one into CHPs or address spaces.

In Figure 1, boxes 1-4 are Cronus process objects; box 5 is the operation switch, which accepts messages from and delivers messages to the Cronus processes on this host; box 6 the local host component of the UNO generation facility; box 7 is the UDP protocol demultiplexing service; box 8 is the IP protocol demultiplexing service; and box 9 is the Virtual Local Network layer.

The operation switch is a table-driven switch, which routes messages from process to process. The sender and receiver may both be on a single host, or the IP layer may be involved in a host-to-host message transfer. The operation switch does not retain information about the messages, although it may gather statistics and transmit them to a central collection point.

Operation switches are linked by a reliable message service. The IPSend and IPReceive discussed below are made reliable through extensions to IP datagram exchange protocols and/or the use of reliable TCP protocols. If an attempted InvokeOnHost fails, the invoker may assume that the problem is not a transient communication fault; with high probability, either the network or the target host, or both, are down. Messages transmitted through IPSend and IPReceive are not limited in size by the maximum packet size supported by the Physical Local Network.

The InvokeOnHost operation is one of the principal system calls to the Operation Switch. The invocation sequence for an operation on another host is:

1. A Cronus process initiates an InvokeOnHost operation, transmitting the operation and its parameters to the operation switch on the source host.
2. The source operation switch composes an IP message containing the object UID, operation, and some other information, and sends it to the operation switch on the target host.
3. The target operation switch uses the object UID and its own tables to decide which process should receive the message, and delivers it.
4. The process on the target host receives the message using the Receive operation; the SourceInternetAddress and SourceProcessUID are those of the invoking process.

If the source and target processes are on the same host, the source and target operation switches are the same, making the

transmission of the data unnecessary.

The following sections explain the function of the Operation Switch in greater detail.

4.4.2 The Operation Switch Interfaces

Figure 2 illustrates the transmission of an operation from the invoking process, through the local operation switch, to the remote operation switch, and finally to the receiving process. This section defines the calls and the representation of data structures at the interfaces 1, 2, and 3.

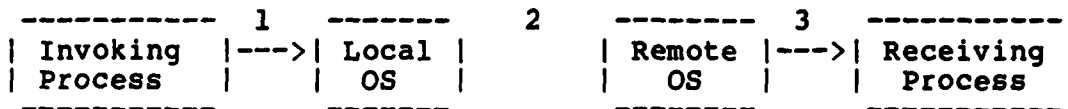


Figure 2 . Operation Switch Interfaces

There are two major views on the invocation in Figure 2: the invoking process may perform a `SendToHost` operation, specifying a destination process name and expect it to be paired with a `Receive` operation at the receiving process; or the invoking process may perform an `InvokeOnHost` operation on the Cronus object name that is ultimately directed to a manager process and again accepted by a matching `Receive`.

In the first case, information crosses interfaces (1) and (3) by means of calls made by the sending and receiving processes; these calls appear as

```
SendToHost(TargetAddress,ProcessUID,MessageText)
```

```
Receive(SourceAddress,SenderUID,MessageText)
```

In the second case, information crosses interfaces (1) and (3) by means of system calls made by the invoking and receiving processes; these calls appear as

```
InvokeOnHost(TargetAddress,ObjectUID,Operation)
```

```
Receive(SourceAddress,SenderUID,ObjectUID,Operation)
```

where operation includes both the intended operation and its parameters.

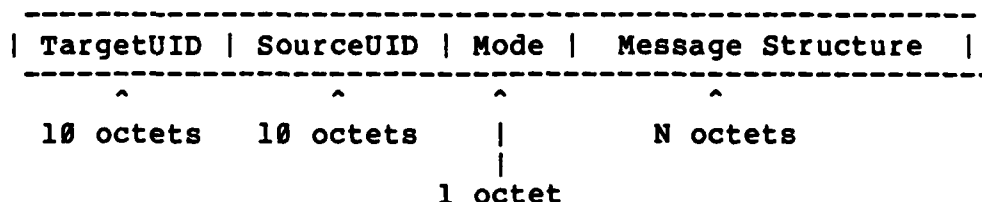
Interface (2) is peer-to-peer communication between operation switches. It is convenient to introduce IPSend and IPReceive for this function. IPSend and IPReceive use a reliable message service built above the Internet Protocol. IPSend and IPReceive each accept just two parameters:

IPSend(TargetInternetAddress,Text)

IPReceive(SourceInternetAddress,Text)

IPSend will perform a certain number of retries in an attempt to deliver a message; IPReceive will filter duplicates arising from retries.

Messages exchanged between operation switches are octet sequences with the following standard format:

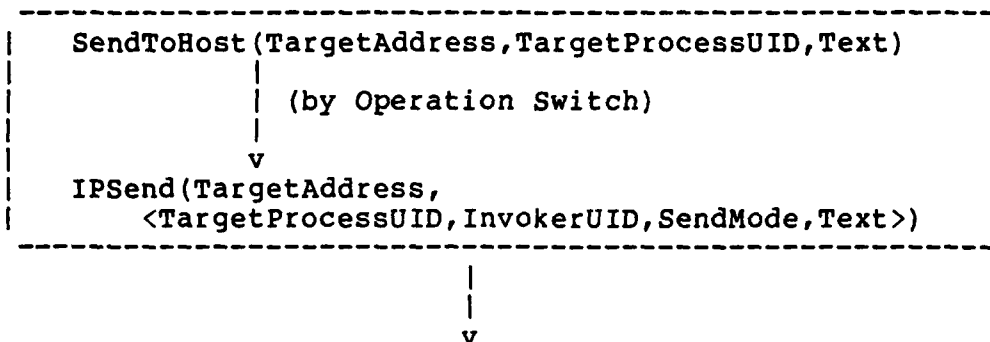


The TargetUID is the ProcessUID parameter to SendToHost or the ObjectUID parameter to InvokeOnHost. The SourceUID is the process UID of the invoking or sending process. Mode is an enumeration variable with two values, SendMode and InvokeMode, explained below. The Message Structure is the MessageText parameter to SendToHost or the Operation parameter to InvokeOnHost.

4.4.3 The Implementation of SendToHost and Receive

The operation switch implements the SendToHost and Receive operations for processes, and assists in the implementation of other operations by directing the messages to manager processes. The first case is illustrated in this section, and the second in the following section.

Host = SourceAddress:



Host = TargetAddress:

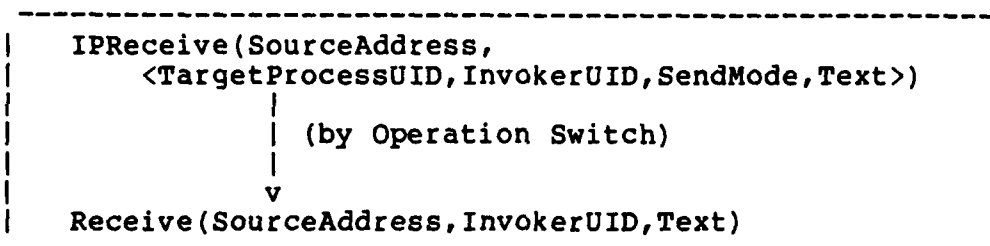


Figure 3 . The SendToHost-Receive Sequence

In Figure 3, the sending process at host SourceAddress initiates the SendToHost operation, and the data passes into the local operation switch. The operation switch at SourceAddress uses IPSend to transmit the data to the Operation Switch on host TargetAddress, where it is received by means of IPReceive. When a matching Receive request made by the target process completes, the SourceAddress, InvokerUID, and Text fields have been made available to the target process (that is, moved into its address space).

The operation switch at the SourceAddress sets the Mode value to indicate that the operation is a "SendToHost". The operation switch at the TargetAddress detects the SendToHost-Receive case by observing that Mode=SendMode; this is sufficient

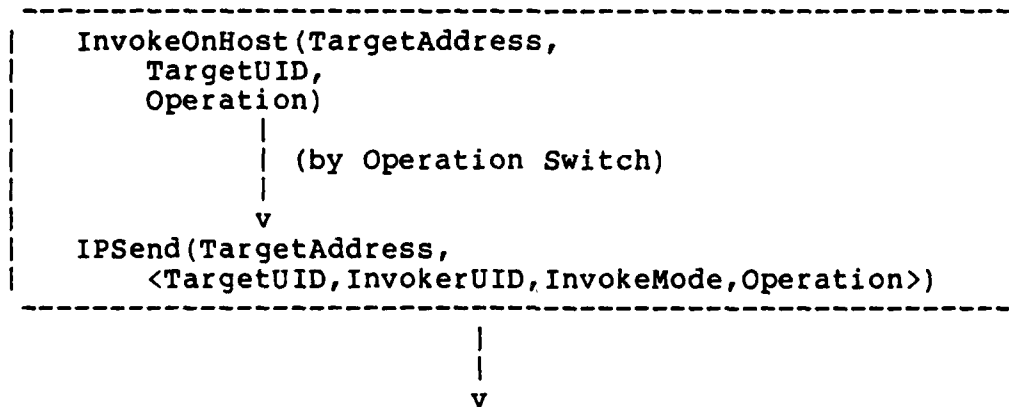
information to complete the matching Receive.

In order to use SendToHost and Receive, an operation switch must know the Cronus UIDs of all processes on its host, and must have a means of passing messages across the Operation-Switch-to-Cronus-process boundary. The operation switch maintains the mapping from UIDs to host-dependent process handles, and uses the host-dependent system call convention to move the data.

The TargetProcessUID may be either a specific or logical UID. If it is a logical name, the target operation switch converts the name to the process UID for the process currently supporting the logical function.

4.4.4 The General Invocation Sequence

Host = SourceAddress:



Host = TargetAddress:

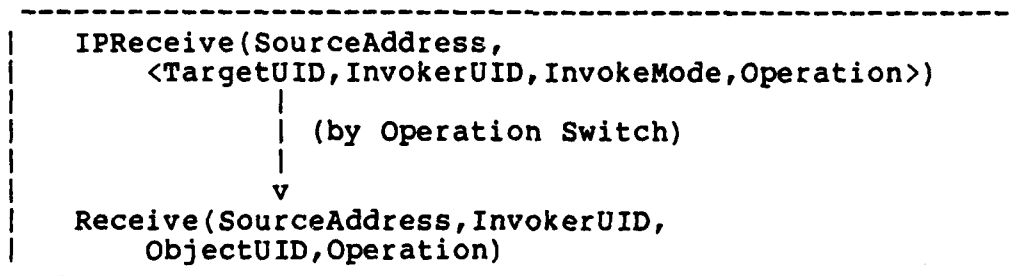


Figure 4 . The General Invocation Sequence

In Figure 4 the invoking process at SourceAddress initiates the `InvokeOnHost` operation, and the data passes into the local operation switch. The local operation switch sets Mode to `InvokeMode`, and uses `IPSend` to transmit the data to the operation switch on TargetAddress, where it is received by means of `IPReceive`. The operation switch on TargetAddress observes that this message is in `InvokeMode`, and delivers the message to the object manager process for this type, whose name is derived from the target UID.

The operation switch on host TargetAddress tests the Type field of TargetUID. If the message has a generic address (the UID type is CT_Type_Name), the operation switch uses the NameToType function to determine the intended type; otherwise, it uses the value of the Type field. The operation switch tries to map the type to a manager process on this host (there is at most one manager process on a host for any type). If the mapping is not successful, the invocation is discarded, but will generate an exception reply. If the mapping is successful, the information is transmitted to the manager process as shown in Figure 4. The manager obtains the information by initiating an ordinary Receive request; when the Receive completes, the SourceAddress, InvokerUID, ObjectUID and Operation have been made available to the manager process.

4.4.5 The Use of UID Location

The operation switch often avoids the Locate operations shown in the definitions of Invoke and Send by using the host address from the UID parameter when it is a reliable hint to the object's location. In this case, the Invoke or Send can be immediately replaced by an InvokeOnHost or SendToHost. Pseudocode for an Invoke operation incorporating this optimization is:

```
operation Invoke(ObjectUID,Operation) is
  HostAddress := OriginOfUNO(ObjectUID.UNO)
  if not GoodAddressHint(.TypeOf(ObjectUID)) then
    Locate(ObjectUID,HostList,1,TimeoutConstant)
    if HostList(1) = NULL then
      return
    else
      HostAddress := HostList(1)
  InvokeOnHost(HostAddress,ObjectUID,Operation)
```

The predicate GoodAddressHint attests to the trustworthiness of the host address in the object UID.

Primal Process and Primal File are important examples of types for which GoodAddressHint returns true.

5 Process Management

5.1 Cronus Processes

5.1.1 Introduction

Each host and constituent operating system in a Cronus cluster has at least one natural concept of the basic unit of computational activity, the process. More generally, several different kinds of processes are present in each host, fulfilling different roles. In the absence of a distributed operating system, the processes on two heterogeneous hosts are unrelated to each other. The first step towards building systems of cooperating processes is to standardize communication protocols, so that the processes on heterogeneous systems can talk to one another.

Standard communication protocols are only the beginning. In a modern operating system, a process is an object which can be explicitly manipulated by other system and application processes. The operating system makes available a group of process control operations. These operations are invoked by a controlling process on a controlled process, often without the voluntary cooperation, or even the knowledge, of the latter. Examples of process control operations are:

- Create
- Terminate
- Suspend
- Resume
- Set Priority
- Set Access Rights
- Interrupt

Cronus provides uniform process control operations across the heterogeneous hosts in a Cronus cluster. Cronus processes are constructed from constituent host processes (CHPs). The properties of a CHP are defined by the machine architecture and the constituent host operating system (e.g., a UNIX CHP is very different from a CMOS CHP). A Cronus process consists of a CHP plus the Cronus process features. A CHP becomes a Cronus process by functional enhancements that usually do not block or replace the CHP's natural features. Cronus processes then have full access to the resources of constituent hosts within the bounds of access control. Unfortunately, the process control operations which are native to different host operating systems are dissimilar in scope and detail. By adopting uniformity as a

goal, we acknowledge the inevitable mismatches between the Cronus process control operations and the operations available on a particular COS. It is the responsibility of the host integrator to bridge this gap.

Requiring full compliance to process control may be too much of a burden for some hosts. The Cronus design provides flexibility in the degree of integration required of a host. The host integrator decides which Cronus types will be supported by the host.

5.1.2 Cronus Process Types -- Overview

There are two basic Cronus process types, CT Primal Process and CT Program Carrier(7). The type CT_Program Carrier is a subtype of CT_Primal Process. Primal processes and program carriers never migrate; once created, the process remains on the same host until it is destroyed. The host hint in a UID for a primal process or program carrier is thus perfectly reliable.

Every host participating in the system must support a Primal Process Manager (PPM) and primal processes. A primal process which plays a well-defined functional role within the system is called a Cronus service. Cronus services are often object managers for system-defined object types, for example, a Primal File Manager or Program Carrier Manager. A Cronus service is a primal process which has a registered Cronus type (and hence a logical name) drawn from the space of Cronus types. Operations can be invoked on a service and messages sent to a service using its logical name.

In their minimal standard forms, Primal Processes and Primal Process Managers are relatively simple. This keeps the cost of integrating a host into a Cronus cluster low for minimally integrated hosts that can obtain system services from other hosts, but do not provide system services.

Ordinary primal processes lack essential process control functions and other desirable characteristics needed for

(7). Future system versions will introduce additional process types which may be distributed in extent and have special reliability properties.

application programming. The subtype CT_Program Carrier provides an environment tailored to the requirements of application programs. For example, a program carrier can be remotely loaded and started.

Either type of process may make use of some or all of the functions in the Process Support Library (PSL) which provides high level interfaces to many system functions, as well as general purpose utilities for interfacing to and manipulating the Cronus environment. Portability is a major goal for the PSL, so that it can be implemented readily in whole or in part on new host types. The PSL is discussed further in Section 14.5.

5.1.3 The Operations on Objects of Type CT_Primal Process

The set of operations defined on objects of type CT_Primal_Process is:

Locate (ProcessUID) -> HostID

Return the internet address of the host supporting this Primal Process. (This is the standard Locate operation defined for all objects of the system).

SendToHost (HostID,TargetProcessUID,Text) -> ReplyCode

Send a message to a Primal Process; the message is accepted by the Receive operation.

Receive () -> SourceHostID,SourceProcessUID,Text

Accept a message sent to this Primal Process, along with the source identity.

Destroy (ProcessUID) -> ReplyCode

Terminate the activities of the Primal Process and release all resources allocated to it. This operation does not cause the process to terminate cleanly.

Report_Process Descriptor (ProcessUID,SelectionList) -> Process Descriptor

Return the requested key-value pairs from the process descriptor belonging to the target process.

Change_Process Descriptor

(ProcessUID,ModifyList,DeleteList,InsertList) ->
ReplyCode

Insert, delete, or modify key-value pairs in the process descriptor of the target process.

A process may invoke any of these operations on itself as the target object (Receive may be invoked only on the invoking process). A process may send itself messages, destroy itself, or read or change its descriptor in the same way it performs these operations on other objects. Locate, SendToHost, and Receive are described in detail in section 4.4, and will not be discussed further here.

The Destroy operation is invoked on a Primal Process to "destroy" or "kill" the process. It erases all record of the process state from the system and frees any resources dedicated to the process.

A process which is destroyed is not notified of the operation, and has no opportunity to terminate cleanly. Only the resources actually used to implement the Primal Process object can be freed directly; resources held as a result of the computational activity of the process (e.g., locks on remote files) are not freed. Some primal processes may possess dedicated resources, and Destroy disables the process, without releasing the resources.

A reply will be generated to the invoker to indicate that the process has been destroyed. After receiving the reply, the invoker knows that future operation using the specific UID of the destroyed process will not succeed.

A process descriptor is a list of key-value pairs associated with a Cronus process. Some of the values are components of the process state used to implement process control. For example, the pair (Priority,5) would indicate the importance of a process relative to other processes competing resources. Some keys must be present in the list ("required keys"), while others are optional.

All process objects must respond to the required keys in a uniform way. If a process object supports a standard optional key, the process must apply use it in a uniform, system-wide manner. Additional, elective keys may be present in a process descriptor. Their interpretation is not specified by the system, but is entirely the responsibility of the process and the other

processes with which it interacts. Elective keys are chosen not to conflict with required or optional keys.

The required keys for Primal Processes are:

MyUID
MyAGS
IPCEnabled

The key MyUID is placed in the descriptor when a primal process is created, and is never changed thereafter. The value of the (MyUID,value) pair is the specific UID of this primal process, and has type CT_Primal_Process.

The value of the MyAGS is the access group set, used with access control lists to determine access rights to objects at operation invocation time. The principal UID associated with this process is an element of the access group set. The initialization and use of this access control and authentication data is discussed in detail in section 7.

The value of IPCEnabled controls communication through the operation switch. If the value is true, the process can send and receive messages in the normal fashion. If it is false, the process may not send or receive messages, or invoke operations on Cronus objects. This feature can be used as a basic tool for managing access to network resources.

The optional keys for Primal Processes are:

Priority

and others to be named at a later time.

The Report_Process_Descriptor and Change_Process_Descriptor operations permit a process to inspect or modify the descriptor of another process. If several processes invoke Report and Change Process Descriptor operations on another process at the same time, the effect will be as if the operations were processed sequentially, i.e., they are atomic with respect to each other.

Report_Process_Descriptor causes a reply to be generated to the invoker. It may be invoked with a SelectionList requesting specific key-value pairs to be returned, in the reply, or it may

ask for the entire descriptor to be returned. Access control restrictions will limit the set of key-value pairs to be returned. Report_Process_Descriptor is also used as the standard "are you there?" function. The reply is generated, independent of the state of the process.

Change_Process_Descriptor has three arguments, a Modify list, a Delete list, and an Insert list. The Reply shows any discrepancies between the requested changes and the changes actually made. All modifications are made first, followed by all deletions, followed by all insertions. A key-value pair might occur in both the Modify and Insert lists, to guarantee that the pair exists after the operation, whether or not it was present before the operation.

5.1.4 Operations on Objects of type CT_Host

The Primal Process Manager (PPM) implements operations concerning primal processes as a class. Some of these operations may be thought of as operations on the host itself. Because of this, we assign it a type, CT_Host.

A PPM is itself a Primal Process, and the operations in the previous section all apply to it. They are activated by InvokeOnHost applied to the logical name of the PPM.

One of the operations, Destroy, has a special meaning when applied to the PPM on a Cronus host. Because the PPM is the implementer of Primal Processes, destroying the PPM destroys all Cronus processes on the host. This forces a shutdown of the Cronus system on the host.

The operations defined on objects of type CT_Host are:

Cronus_Restart (HostID) -> ReplyCode

Combines the effects of a Destroy operation on the PPM, followed by a "Cronus boot".

Create_Primal_Process (HostID,Role) -> ProcessUID

This operation takes a role designator and starts a Primal Process performing this role on the HostID. The Primal Process is bound to a program when it is created, in a host-dependent way invisible to the Cronus system.

Service_List (HostID) -> ServiceOnHost

Returns a list of the services which can be created on this host, and indicates which are currently active.

Process_List (HostID) -> ProcessOnHost

Returns a list of the specific UUIDs of all active Primal Processes on this host.

Status_Probe (HostID) -> StatusDescriptor

Returns a list of key-value pairs giving information about the current status of the host-device utilization, number of active processes, etc.

Create_Primal_Process takes a role designator as an argument, and starts a new primal process performing this role. The role designator may be in one of the following forms:

1. A Cronus logical name for the service.
2. A Cronus symbolic service name. These are character strings containing the literal characters of a logical name, for example "CL_Primal_File".
3. A host dependent role designator. These are arbitrary strings, which have meaning only to the PPM on a specific host.

The designators of kinds (1) and (2) are strictly paired, and are registered with the Cronus system administrator. They are the names of standard Cronus functional units, which have unambiguous meaning system-wide. The primal processes which implement them are created using a designator of kind (1) or (2), which makes the logical name known to the operation switch on the host, so that the process can be generically addressed.

Designators of kind (3) provide for the activation of host-specific programs or devices. The host dependent role designator might be a COS-dependent file that is executed as a result of Create_Primal_Process. Primal processes created with a host-dependent role designator generally have no associated logical name, and cannot be generically addressed.

When the primal process is created, it receives a new specific UUID, never before used to name a Cronus object. The

primal process will initialize its state entirely from non-volatile storage (local or remote disks). The PPM will generate a reply to the invoker indicating success or failure of the operation; if it was successful, the reply will contain the specific UID of the new process.

5.2 Program Carrier

5.2.1 Objects of Type CT_Program_Carrier

The type CT_Program_Carrier is a subtype of CT_Primal_Process, and all of the characteristics of primal processes are inherited by program carriers. Additional operations can be invoked on program carrier objects, and the set of required keys in the process descriptor is enlarged. The program carrier

- o provides a process which can be created, loaded with a program, started, and stopped under remote control;
- o binds processes to their principals;
- o provides uniform monitoring and debugging support; and
- o provides application developers with the ability to control a collection of user written (possibly distributed) processes.

A Cronus host is not required to support the CT_Program_Carrier process type; however, hosts which are not dedicated to system service roles usually support Program Carriers.

5.2.2 Operations on Objects of Type CT_Program_Carrier

The set of operations defined on objects of type CT_Program_Carrier include those of its supertype, CT_Primal_Process, and:

Clear_Program (ProcessUID) -> ReplyCode

Stop the process cleanly, if it is running, and clear all program and data storage private to the process.

Load_Program (ProcessUID,ProgramUID) -> ReplyCode

Load a binary program image into the process; the process must be in cleared state.

Proceed (ProcessUID) -> ReplyCode

Start execution after a program load, suspend operation, breakpoint halt, or single step.

Suspend (ProcessUID) -> ReplyCode

Stop the process as soon as possible, and save sufficient state to permit a restart when the Proceed operation is invoked.

Stop (ProcessUID,StopCode) -> ReplyCode

Terminate this program carrier process cleanly, according to StopCode procedures and inform the Controller of this process that it has been destroyed.

Report_State(ProcessUID) -> ProcessState

Inspect process private state information.

Change_State (ProcessUID,ProcessState) -> ReplyCode

Change process private state information.

Breakpoint (ProcessUID, Address,InsertOrDelete) -> ReplyCode

Place or remove a breakpoint in the address space of the process.

These operations are sufficient to meet two basic objectives: 1) It is possible to load a binary image into a new program carrier object, start it, and allow the process to complete or be cleanly stopped; and 2) the Suspend, Proceed, Report_State_Change State, and Breakpoint operations together with the Primal Process operations, will support the operation of a remote debugger.

Report_Process_Descriptor and Change_Process_Descriptor can be applied to program carrier objects as well as primal process objects. The required keys for program carriers are:

MyUID
MyAGS
IPCEnabled
Priority
State
Standard_Input
Standard_Output
Controller
Session
Attendant
Current Directory

MyUID, MyAGS, IPCEnabled, and Priority have the same meaning for program carriers as for primal processes.

The State variable informs other processes of the current state or mode of a process. The set of states includes Clear, ReadyToStart, Running, Suspended, and DebugWait. The states reflect only the interactions of Cronus operations and the process object, and do not capture finer state subdivisions which are host or local operating system dependent.

The Standard_Input and Standard_Output keys each has the handle for a data stream as a value. These streams are initialized before a program carrier is started. The streams are used in a manner analogous to the standard input and standard output of the UNIX process model.

The value of the Controller key is a Cronus process UID, the controlling process. Program carrier processes exist in a controller-controllee hierarchy; each controller may have many controllees, but each controllee has one controller.

The Session key is a UID identifying the user session in which this process was created. This value can be used to identify the processes belonging to a terminal session.

The Attendant key may have either a null value, meaning the process is currently unattended, or the UID of the terminal agent of a logged-in user. The Attendant may be used as the target for error messages which should be presented to a human being, providing a standard error channel.

The optional keys for program carriers are:

Program_Name

Program_Load_File
Program_Version
User_Environment

and others to be defined later.

5.2.3 The Program Carrier Manager Operations

The operations which may be invoked on a Program Carrier Manager are:

Create_Program_Carrier (HostUID,Controller) -> UID

Create a new program carrier process in Clear state, and return the UID of the process to the invoker.

Resource_Test (HostUID,ResourceTestList) -> ReplyCode

The parameters are the host resources needed for the execution of a particular program, e.g., memory requirements; the reply indicates whether or not they are available.

Search_All_Descriptors (HostID,FieldList) -> List of UIDs

The parameter is a set of key-value pairs; the reply contains the UIDS of all program carriers on this host which contain all of the key-value pairs in their descriptors.

Create_Program_Carrier creates a new CT_Program_Carrier process and returns the UID of the new process. The new process inherits the process descriptor of the creator, except for MyUID, which becomes the UID of the new process; the streams Standard_Input and Standard_Output, which are unbound; and potentially the Controller entry. The new process inherits the AGS, and hence the authorities, of the parent process.

A (optional) parameter allows the creator to specify whether the new process should inherit the controller of the parent, or should receive the parent's UID as its controller. When the process is created, a message is sent to the controller containing the parent and child process UIDs. The controller uses these messages to keep track of the processes it controls.

A group of routines will be available through the PSL to carry out the standard bookkeeping operations.

Once a process has been created, the parent (or another process) may alter values in its process descriptor, by means of the `Change_Process_Descriptor` operation. The parent may use this operation to change the Attendant or Controller values, or to establish bindings to `Standard_Input` and `Standard_Output`. Support for I/O redirection of the Standard Input and Standard Output streams is provided through routines in the PSL.

The `Resource_Test` operation allows a process to test for the availability of resources before performing the `Create_Program_Carrier` operation. Resources may include processor type, primary memory size, and special processor capabilities, such as floating point hardware. This operation is used as part of the scenario for selecting a site at which to run a program (see Section 13.8). The current design does not support resource reservation.

The `Search_All_Descriptors` operation allows the invoker to find all program carrier processes on a host with the designated key-value pairs in their descriptors. Two important uses of this operation are: 1) a search on the Session key-value pair, to locate all process associated with a user session; 2) a search on the Attendant key-value pair, to locate all processes currently attended by the same terminal device process.

5.2.4 Bindings Between Processes

The Cronus Process Structure supports several kinds of relationships among processes. All processes belonging to a session are related, and can be located as a group; pairs of processes are related in controller-controllee relationships; and processes are bound together by the data streams that connect `Standard_Input` and `Standard_Output` (and by other streams that may be explicitly opened by the processes).

The knowledge that a group of processes belong to the same session is useful for coarse-grained error recovery (killing the session). Streams are used primarily to provide continuous data paths between processes.

The controller-controllee relationship supports the flow of control information among processes. When a process is destroyed, a message is sent to its controller. The controller can then use that information to notify or terminate other controllees that were communicating with the first process.

6 Interprocess Communication

6.1 Overview

The message oriented interprocess communication (IPC) facility uses the primitives `SendToHost`, `InvokeOnHost`, and `Receive` (see Section 4). This facility supports both the system implementation and application program needs for efficient control message communication. There are further requirements for supporting IPC in Cronus. First, there is a need to adopt conventions for the common interpretation of the messages. These conventions govern both the form of the message and its content. Second, in the network environment IPC is found in two general varieties, control messages and streams. Both modes of IPC are useful and natural to different programming needs. In this section we discuss the design of message structure conventions and higher level IPC abstractions.

6.2 Message Structure

6.2.1 Objectives

The Cronus message structure design assumes that the dominant goal is the regularization of control traffic in the heterogeneous Cronus system. Control traffic includes but is not limited to requests for operations to be performed on objects, replies generated by operations, exception notices, and messages needed to coordinate distributed object managers. Control messages are usually short (tens to hundreds of octets). Because control messages are often in the critical path to completion of an interactive command, performance is a major issue. Messages should be compact, and efficiently composed and parsed.

The Cronus message structure conventions are realized by a group of software components collectively called the Message Structure Facility. The Message Structure Library (MSL) is the implementation of an MSF component, a library of functions or procedures which are available to processes on every Cronus host. Messages are composed by passing information to the MSL procedures; the result of a sequence of such calls is a data structure in the Standard External Representation (SER). This data structure can be transmitted from one process to another, and subsequently parsed by MSL procedures at the receiving process.

The objectives for the Cronus message structuring facility, in approximate order of importance, are:

1. Lossless Storage. A process must be able to extract all of the information inserted into a message structure by the process which created it.
2. Performance. The message data structure must be compact.
3. Portability. The MSL implementation should be easily portable among the hosts in the Cronus ADM.

Attaining Objective (1) assures us that the MSF can be used to move an arbitrary data structure (viewed as a bit- or octet-vector) from one Cronus host to another. The representations of the data structures may differ at the sending and receiving hosts, but no information will be lost. For example, on the VAX a message in the SER may be stored as a consecutive sequence of 8 bit bytes, while on the C/70 the same message is stored as a sequence of 10 bit bytes.

The MSL contains the functions to handle machine dependent conversions to standard data representations. An example would be a MSL procedure on the C/70 which coerces a 20-bit C/70 integer into a 16-bit standard SER integer; some of the dynamic range of the C/70 integer is lost in the conversion. These procedures define relationships between SER data structures and machine- or language-specific data structures, and are inherently non-portable across heterogeneous machines and/or language systems.

Portability of the MSL, Objective (3), helps reduce the cost of the MSF implementation on the eight or more hosts in the ADM. Large portions of the MSL will be portable among all of the ADM hosts supporting the C language, with no changes to the MSL source files.

6.2.2 Message Structure Conventions

6.2.2.1 Self-Description

A message is self-describing if it contains information about its own structure, or about the structure or type of its components. A convention for message structure is self-describing, if every message which conforms to the convention contains some self-descriptive information. A receiver can depend upon the presence of this information, and need not rely upon higher-level protocols for its inclusion.

For example, a receiver might expect a message containing a timestamp; a timestamp might be represented either as a binary integer of 32 or 64 bits, or as a fixed length ASCII string. If messages contain no self-descriptive information, the receiver must make prior arrangement with the sender to either: 1) place exactly one of the possible formats (e.g., 32-bit binary) in every message; or 2) indicate in each message which variety of timestamp was included. In case (2) the question of self-description recurs, over the representation of the indicator field.

The Cronus conventions for message structure contain self-descriptive information.

6.2.2.2 Language Integration

Conventions for message structure can be influenced by the programming language compiler and machine architecture used to implement them. Tight integration would be achieved by developing a representation for a linguistic structure such as a Pascal record or C structure and enforcing conformance by compilers used; integration is achieved by packages which strive for portability, and must be compiler-, language- and machine-independent to a large degree.

Tight integration improves performance, because the compiler can optimize reference to messages. The burden for defining the data types of message fields can be borne by the typing facilities of the host. On the other hand, tight integration implies a strong dependence on a single language and compiler, omitting or building distinctly less well-integrated packages for other language environments.

The weakest form of integration implies reliance on a few

language features that are present in many languages. For example, the library of routines might use procedure calls as the only form of invocation, arrays of integers as the only data structure, and only stack-oriented storage allocation at procedure entry time. A library which obeys these constraints could be easily implemented in Pascal, C, PL/1, and most other block-structured languages. If portability is an important goal, the implementation can be made portable across a range of compilers and host machines. Thus weak integration allows structure conventions to be implemented uniformly on many systems, at reasonable cost.

The Cronus environment requires us to refrain from using a single language base. In that sense, the conventions employ weak integration. However, as a practical matter, we are attempting to limit our development activities to a single language to enhance portability within the ADM and to minimize the effort in bringing up initial components.

6.2.2.3 Data Type Support

A language-based convention will generally permit messages to contain some or all of the standard types defined in the language. In the simplest case, a convention may consider messages to be composed only of bit- or byte-strings. The responsibility for interpreting the message fields as integers, character strings, etc., is left to higher-level software. A somewhat more complex convention may define the representations of basic data types (e.g., integers, booleans, and strings) in a language- or host-independent way. These data types may or may not include composite types (e.g., lists, records, arrays) which can be used to build complex message structures.

A convention may explicitly acknowledge the ability of users to define new types, to be treated like the pre-defined types. There may be an administrative authority responsible for guaranteeing the uniform interpretation of the types which evolve after the convention has been established.

If the application domain is well understood, the convention may incorporate data types especially important to the domain. Control data might be a set of these conventional types, for example, Universal Identifiers, Transaction Identifiers, and timestamps in various formats.

The issue of specification of data type representation is separable from the issue of self-description. A convention which specifies the representation of a 32-bit, two's complement integer, for example, may or may not include a type tag on elements when they are embedded in a message structure.

The Cronus message structure conventions are layered; the lower layer assumes only integer and sequence of integer data types, and is thus highly portable. The upper layer defines a group of data types (e.g, integers and strings) in terms of the sequence of integer type, which can effectively handle host heterogeneity but are not portable.

6.2.2.4 Performance

Execution time costs associated with message structures can be roughly divided into three categories:

1. Since messages are transmitted through the IPC facility, the cost of transmission accepting the message into the IPC facility, buffering, transmitting, receiving, buffering, and finally delivering the message to a client) is an increasing function of message size.
2. Message structures are composed and accessed by routines which implement the conventions the cost of these operations is a function of the complexity of the conventions.
3. The semantics of an application will not mesh perfectly with the data typing or structural concepts provided by the convention; there is a cost (borne by the clients) for encoding higher-level concepts in those known to the convention.

If a convention is insufficiently rich in concept, (3) may be the dominant cost of use. If it is too complex, (2) may dominate. The most desirable situation is one in which (1) dominates, and furthermore most of the information content of messages is useful to the recipients.

6.2.3 The Cronus Message Structure Facility

The Cronus MSF uses an external representation based on key-value lists, where the key stored with each data value indicates the meaning of the value. Both keys and values can vary in length from one octet to many thousands, and are not restricted in form. The data structure built by the MSL functions will have a unique value for each key present in the structure. Null values are possible, and often useful when the presence or absence of an key is an adequate expression of intent. The type and structure of each value field is encoded along with the value.

The assumption that most messages encoded in the standard external representation are small has an important consequence: small messages have little substructure. Because the average key or value is small and lacking in substructure the SER does not explicitly encode recursive data structures, for example, values which are themselves key-value lists.

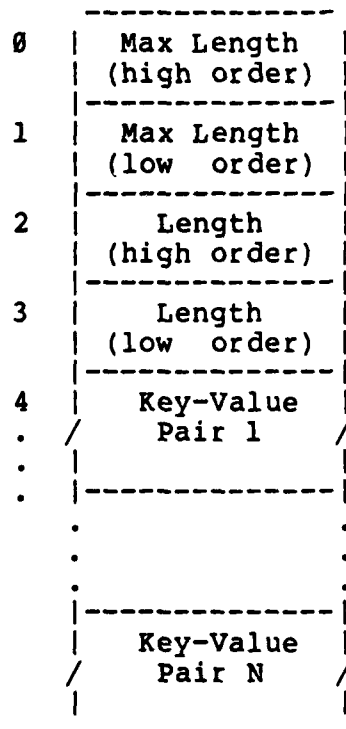
The case in which a value is a list of similar elements is common enough to warrant special attention. The MSL contains functions which treat the value part of a key-value pair as an array of fixed-sized elements.

The MSF-to-client interface is defined by the functions (i.e., entry points) of the MSL. The functions are divided into two classes, host independent and host dependent. The host-independent functions of the MSL rely on the language environment to support simple data structures (integers and sequences of integers); the host dependent functions provide conversions between complex, host dependent data structures (e.g., floating point quantities) and the SER.

6.2.4 The Standard External Representation

The Standard External Representation defines a data structure and a transmission order for the octets in the structure. Knowledge of the representation is sufficient to construct and interpret data structures exchanged among system components through the Cronus local network.

A SER data structure is a sequence of octets, where each octet is considered as a small integer in the memory of a host. It begins with 4 octets of header information, followed by a list of key-value pairs, as shown in Figure 5.



One or more pad octets may occur before the first key, between a key and its value, or after the last value. Pad octets have the bit pattern '00111111'. Pad octets are skipped over when the message structure is parsed.

The Max Length field of a SER data structure represents an integer in the range 0..65535. The Max Length field is set by the client when composing the message; the MSL routines will not permit the length of the resultant structure to exceed Max Length. The current length, in octets, of a SER data structure is stored in the Length field (octets 2 and 3). The integer values of the Max Length and Length fields are computed as (high-order-octet)*256 + (low-order-octet).

A key-value pair consists of a key specifier followed by a value specifier (possibly separated by pad octets). Both

specifiers have the same format, and each may be as short as one octet or a maximum of 16,385 octets. The first octet of a specifier designates one of four possible encodings, as a function of the two high-order bits in the octet. Figure 6 illustrates the possibilities.

To avoid ambiguities, it is necessary to restrict the way arguments to MSL functions are encoded in the message. Assume that a key or value argument of length N octets is to be inserted in the message:

1. A one-octet key or value in the range 0..62 is encoded in a Type 0 specifier.
2. A two-octet key or value in the range 0..16,383 is encoded in a Type 1 specifier.
3. All other keys or values are encoded in Type 2 or 3 specifiers, in N+1 or N+2 octets, respectively.

These rules guarantee that the key or value and its length can be recovered from a SER structure unambiguously.

In this format, small integers are encoded in either one or two octets, as a function of the required range. Short ASCII strings are encoded with an overhead of one octet; large data blocks (e.g., pages read from a remote disk) are encoded with an overhead of two octets. The MSL functions select specifier types automatically, on the basis of their arguments. When MSL functions are used to build and parse message data structures, the presence of four specifier formats in the SER is invisible to the client.

6.2.5 Canonical Types

A canonical type defines a standard representation for values of a basic data type in a sequence of octets. The use of canonical types in message structures is by convention between communicating processes, but it is usually the case that the key and/or value of a key-value pair to be represented as an element in the value set of a canonical type.

Type 0

Specifier is 1 octet long, the value is xxxxxx,
in the range 0..62:

```
-----
| 00xxxxxx |
-----
```

Type 1

Specifier is 2 octets long, the value is
(xxxxxx*256)+yyyyyyyy, in the range 0..16,383:

```
-----
| 01xxxxxx | yyyyyyyy |
-----
```

Type 2

Specifier is 1 to 64 octets long, xxxxxx is the
number of value octets after the first octet of
the specifier:

```
-----/-----
| 10xxxxxx | dddddddd | ... | dddddddd |
-----/-----
```

Type 3

Specifier is 2 to 16,385 octets long,
(xxxxxx*256)+yyyyyyyy is the number of value
octets after the first two octets of the
specifier:

```
-----/-----
| 11xxxxxx | yyyyyyyy | dddddddd | ... | dddddddd |
-----/-----
```

Figure 6 . Specifiers for Keys and Values

The octet can be viewed as the primitive canonical type. An octet can be manipulated as an unsigned integer in the range 0..255 on any host supporting the MSF. The remaining standardized canonical types, listed below, are represented in one or more octets.

<u>Abbrev</u>	<u>Data Type</u>	<u>Length (Octets)</u>
BOOL	Boolean	1
U16I	Unsigned 16 Bit Integer	2
S16I	Signed 16 Bit Integer	2
U32I	Unsigned 32 Bit Integer	4
S32I	Signed 32 Bit Integer	4
ASC	ASCII String	1 or more
BITS	Bitstring	2 or more
UID	Universal Identifier	10

The boolean type encodes "true" as 1 and "false" as 0.

The integer types place the high-order bits of the integer representation in the octet with the smallest index. Signed integers are represented in two's complement.

Strings consist of a variable number of octets, each octet representing an arbitrary character from the full ASCII character set. An ASCII character is stored as a small integer in the range 0..127, i.e., the high-order bit of each ASC octet is zero.

A Bitstring of N bits ($N > 0$) is represented by a sequence of $((N-1)/8)+2$ octets, consisting of a prefix octet followed by a sequence of data octets. The prefix octet contains an integer in the range 1..8, specifying the number of valid data bits in the last data octet.

A Universal Identifier type is defined as a canonical type because UID's will occur frequently in messages, and it is convenient to develop standard functions to manipulate them. The functions will transform a UID from a host dependent representation to the canonical representation, and the reverse.

Additional canonical types will be defined as the need arises.

6.3 Higher Levels of Interprocess Communication

The operation switch provides an object-oriented IPC mechanism with the primitives InvokeOnHost, SendToHost, Send and Receive, which operate on messages.

This section describes additional layers of the IPC function, which go beyond simple message communication in a number of important respects:

1. Asynchrony and demultiplexing: Processes may engage in many simultaneous transactions. There is a need for asynchronous message delivery and a facility that matches incoming messages with the appropriate processing for them.
2. Transactions among cooperating process: There are a number of important message exchange paradigms used to support the Cronus functionality.
3. Streams: The stream concept (a unidirectional or half-duplex connection) is introduced.
4. Stream redirection. When a source and a sink process both regard data flow to be taking place over a stream, the effect is similar to a UNIX pipe. To make streams more useful the logical binding of two processes to a stream can be established by a third party process.

In the rest of this section we discuss facilities for coordinating the use of message exchange and the development of a stream concept. These discussions are very brief because the design is currently being worked on. We include this tentative discussion of the design to indicate its direction.

Receiving data is more complex than sending it, because 1) a receiver may wait a long time before receiving data from a particular source, and 2) it must be prepared to process data from other sources in the meantime. Demultiplexing the incoming message sequence is the responsibility of routines in the Process Support Library. These routines are concerned with:

1. Demultiplexing: Data received from different origins is sorted according to various criteria, so that the receiving process can react properly.
2. Asynchronous receives: The receiver must be able to

compute while waiting for IPC data. The computation may include processing other IPC data, or it may be unrelated to the communication activity of the process.

3. Buffering: Incoming data should be buffered so that the sender is not delayed. The receiving process should be able to control the commitment of resources to buffering.

The demultiplexing facility uses well-specified key, value pairs in the message structure to decide how an incoming message is to be processed. In addition, the key, value pairs transmitted form the basis for a message oriented transaction protocol which organizes the cooperative behavior of multiple Cronus processes.

6.3.1 Message Patterns

Messages which support remote operations are of four types: Request, Reply, Handoff, and InProgress. The simplest (and probably most common) case involves one Request message generated by the invoker, and one Reply generated by an object manager in response. This case is diagramed in Figure 7.

A manager may delegate some or all of the responsibility for performing an operation, and the communication protocol contains the Handoff message type for this purpose. Figure 8 shows a request sent to a manager process and then "handed-off" twice to other manager processes before a reply is transmitted to the invoker. Any number of handoffs may occur between the request and reply messages; the processes which handle the message may transform the message data structures in arbitrary ways before the next handoff or the final reply message is sent. Thus in the event that an operation is handed-off several times, the last manager process to receive the operation may "perform" it, or the manager processes which handle the operation may share the burden, each performing a specialized sub-operation.

Figure 8 also illustrates the InProgress message type. During manager1's handling of the request, manager1 may send an InProgress message to the original requestor. Any number of InProgress messages may be generated by manager processes handling a request; they are all addressed to the process which initiated the Request message (8).

(8). In Figure 8, manager1 would address the InProgress message

At Invoking Process

At Manager Process

```

-----
.
.
.
{ prepare parameters }
Invoke(targetUID,...)
      ---- Request Message ---->
                                Receive(invokerUID,targetUID,...)
                                { perform the operation }
                                Send(invokerUID,...)
      <---- Reply Message ----
Receive(managerUID,...)
{ interpret reply }
.
.
.

```

Figure 7 . A Two Process Invocation (pseudo-code)

All of the messages in the operation protocol are marked as belonging to the operation protocol, and each is marked with its type--Request, Reply, Handoff, or InProgress. All messages arising from one Request contain the same Cronus unique number called the operation identifier; Messages arising from different Request messages contain different operation identifiers. A Request message also contains the operation name; a Handoff message contains the operation name from the Request message, and a "reply to" field; and a Reply message contains a standard reply code. These are the minimal contents of the messages; they also contain additional, operation-specific information.

We distinguish between a simple operation (or operation) and a compound operation. The preceding paragraphs describe the operation protocol as it applies to a simple operation. A simple operation has, by definition, a single operation name and operation ID. When a simple operation is handed off, it retains

to the source process of the Request message; manager2 would address the InProgress message to the process named by the value of a reply-to message field.

<u>At Invoking Process</u>	<u>At Manager Processes</u>
.	.
.	.
.	.
{ prepare parameters }	
Invoke(targetUID,...)	
----- Request Message ----->	
	manager1: Receive(invokerUID,targetUID,...)
	{ perform part of first subtask }
	<----- InProgress Message -----
Receive(manager1UID,...)	.
{ interpret progress report }	.
	.
	{ perform rest of firstsubtask }
	Send(manager2UID,...)
	----- Handoff Message ----->
	manager2: Receive(manager1UID,targetUID,...)
	{ perform part of second subtask }
	<----- InProgress Message -----
Receive(manager2UID,...)	.
{ interpret progress report }	.
	.
	{ perform rest of second subtask }
	Send(manager3UID,...)
	----- Handoff Message ----->
	manager3: Receive(manager2UID,targetUID,...)
	{ complete the operation }
	Send(invokerUID,...)
	<----- Reply Message -----
Receive(manager3UID,...)	
{ interpret reply }	
.	
.	
.	

Figure 8 . A Multiple Process Invocation (pseudo-code)

its identity--that is to say, it retains the original operation name and operation ID.

Any manager process, in the course of acting upon a Request or Handoff message, may invoke one or more new (simple) operations by sending Request messages. A compound operation is the aggregate of all simple operations arising from or caused by the invocation of one simple operation. Normally, all of the "suboperations" will complete before the initiating simple operation completes. A compound operation may be a simple operation or it may be composed of many simple operations, in general with different operation names. Each of the simple operations has its own unique operation ID; if this were not the case, a process that invoked several sub-operations in parallel might be unable to associate replies with invocations.

It is desirable for a Cronus process to be able to query the status of a compound operation. The process initiating a compound operation has immediate knowledge only of the operation ID of the initiating simple operation. By transmitting this ID in the Request and Handoff messages of all simple operations it causes, the managers acting on suboperations have enough information to respond to a status query keyed to the initiating ID.

6.3.2 Stream IPC

A Cronus stream is a uni-directional data channel between two Cronus objects. It has a source object that produces data and a sink object that consumes data. Streams will be used to interconnect processes with files, devices and other processes. A source or sink object may be a static object such as a file. Ultimately, however, the static object is represented by a process which is one end of the stream.

Data flows only from the source to the sink. However, the implementation of a stream involves transmissions in both directions: from source to sink containing data, and from the sink to source containing flow control and synchronization information. There are a variety of implementation techniques which can be used to support the stream concept. These include inter-host TCP connections, the exchange of multiple messages; or even a locally supported mechanism such a UNIX pipe, when the entities are co-located.

The stream concept is supported by:

1. Library routines in the PSL, which provide the stream interface to system and application programs.
2. The object-operation protocol and the transmission of large messages.
3. A message stream protocol, implemented by the PSL library routines.
4. Other existing host support software such as TCP connections.

7 Authentication, Access Control, and Security

7.1 Introduction

The goals of the Authentication and Access Control facility are:

1. Prevention of unauthorized use of Cronus and unauthorized access to DOS maintained data and services.
2. Preservation of the integrity of the system and its components against intentional insertion of unauthorized components.
3. Support for a uniform user view of access control to the resources and functions provided by Cronus.

The design of the access control and authentication facility assumes that systems in a Cronus cluster are all in a single administrative domain. There are three broad classes of hosts within the cluster:

- o hosts dedicated entirely to Cronus system functions and not user programmable;
- o hosts supporting user applications using tamper-proof multiple protection domains (trusted multi-access hosts); and
- o hosts supporting user applications without secure multiple protection domains (single-user workstation hosts).

We assume all hosts supporting dedicated Cronus functions and multiple user protection domains are physically secure from tampering. Workstations may not be completely physically secure, but have at least a tamper-proof component. At minimum, this component is in the local network address insertion and reception function. It could, however, be higher up in the workstation system: in the virtual local network internet address insertion and reception function; in the object system process-unique identifier insertion and reception function; or even higher. In this sense, all user-programmable hosts support multiple protection domains (user and system), although in the limiting case, the "system" domain may simply be a piece of network interface hardware. Since we are not aware of any workstation systems meeting this requirement, we assume future product

packaging changes. There seem to be two viable positions to take regarding the assumptions on these changes.

1. Assume only an absolute minimum, that a single low level "address" can be protected.
2. Allow the set of protected functions to grow as needed to conveniently interface the workstation in a manner as similar as possible to multi-access systems.

The extreme solution to the second approach could be an access machine for each workstation, although other solutions are also possible. For our current work we will assume the second approach, planning only for an arguably insecure implementation directly within the workstation.

The network (cable) itself may also not be totally physically secure. While parts of it can be expected to be secure (e.g. within a secure machine room), other parts can be expected to be exposed to unauthorized connection.

7.2 The Cronus Access Control Concept

7.2.1 Decomposition of the Access Control Problem

The basis of access control in Cronus is the ability of Cronus to reliably deliver the address of a sender of a message (or invoker of an operation) to the receiver of the message. The Cronus communication subsystem is implemented so that this is true. That is:

for IP and Virtual Local Network:

If the sender is within the Cronus cluster, the internet host address of the sender is reliably delivered to the receiver. If the sender is not within the cluster, a non-cluster internet host address is delivered to the receiver, which can be interpreted by the receiver as indication that the authenticity of the sender's address might be suspect.

for the Cronus IPC/object system:

The UID of the sending or invoking process is reliably delivered to the recipient of the message.

The recipient of a request can decide on the basis of the sender's identity whether or not to perform an operation requested.

For this to be a useful basis for access control, a means for reliably associating some authorization with senders' addresses and process UIDs is required.

One approach is to make static bindings between authorizations and addresses or UIDs. These bindings would be "well-known", such that when a process receives a request from the process with UID_Y it knows that the process is acting under the Z_Authority. This method is used in the ARPANET TELNET and FTP protocols; users assume that the process for sockets one and three under the authority of the host administration and can be trusted with their passwords. Static bindings are too restrictive to be the sole mechanism in a system like Cronus, although a few static bindings are required for the access control mechanism to work (see Section 7.6).

Dynamic binding is useful when authorities are not all known at system creation time, and when processes are dynamically created. The system must not only support mechanisms to dynamically establish the binding between a process and an authority, but also to dynamically determine the binding from some system entity in a trustworthy manner.

Most Cronus activity is the result of requests initiated by users of the system. Human users are represented by an abstraction called a "principal". If we extend the notion of a principal to include elements of the system, such as object managers, all activity in the system can be thought of as initiated by principals. System elements which are principals are called "system principals". Each Cronus principal (human or system entity) has a unique identifier. Different system principals have different authorities. For example the primal file manager and the printer service are Cronus system principals, neither of which need be authorized for all of the objects and operations accessible to the other.

Access control can be thought of as consisting of the following steps:

1. Identification. Determine the identity of the principal that is requesting a particular operation.
2. Authorization. Determine whether the principal has been authorized to perform the operation.

For example, when an object manager must decide whether to perform an operation, it must know the identity of the principal that is requesting the operation (Identification) and the rights the principal may have with respect to the operation (Authorization).

7.2.2 Authorization

Cronus uses access control lists to support authorization. The access control list (ACL), which is part of the object descriptor, "protects" a particular action. In the simplest case, it is a list of the principals who have authorization to perform the action. When a principal attempts an operation, the list is checked for the principal; if the principal is present the authority to perform the operation has been verified and the operation may occur.

In Cronus this simple idea is extended in two ways:

1. Group identifiers may appear on an ACL, so an entire group of principals can be authorized as a unit, or have its authorization revoked as a unit.
2. A set of rights is associated with each identifier on an ACL. A single list can selectively control a principal's or a group's access to an object for which several operations are defined, such as a file. There is a right corresponding to each possible operation.

An ACL is a list which contains elements of the form:
(id, rights)

where "id" is either a principal (PID) or a group identifier (GID), and "rights" define the principal's or group's authorization with respect to the object the ACL protects. The allowable rights for a particular ACL are dependent upon the kind of object being protected.

Users log into Cronus as principals by supplying the appropriate name and password(9). A system component called the

(9). See section 13.3 for a more complete description of the login and session initiation scenarios.

Authentication Manager maintains records of all principals and groups. Collectively, these records form a User Data Base (UDB). At login time the Authentication Manager expands the membership of a user-specified subset of the access control groups which he is a member. This is a transitive closure computation on the specified list of group identifiers in the user's record. The user's own id, PID, is added to the result of the expansion. The resulting set of principals is called the access group set (AGS) for the process:(10)

AGS = {PID} U Show_Group_Membership_Expanded (GID)
for the default GIDs in the PID record.

The AGS is used in access control checks as follows. When an action protected by an ACL is attempted, the ACL is compared with the principal's AGS. If an entry of the form:

(ID, (... , Right, ...))

where

ID is in AGS, and
Right is required to perform the action

is found on the ACL, the principal's authorization is verified and the action may be performed.

During a session, a user can remove identities from the current AGS. Group identities may be added, provided the user is a member of the added group. This is accomplished by operations invoked on the Authentication Manager, which causes the update of the current process AGS list. These operations affect a single process only.

7.2.3 Identification in Cronus

There are two related identification problems:

11. At the start of each session, the identity of the user must be established.

(10) The basic ideas associated with Access Group Sets have been adapted from similar work at Carnegie Mellon University in the Central File System project.

12. Processes must be able to ascertain the identity of the principal corresponding to the processes with which they interact.

The solution to both problems lies in a set of mechanisms that bind processes with principal aids and group identifiers. These mechanisms depend upon the ability of the communication system to deliver the UID of a sending process to the receiver of a message reliably.

It is useful to restate these problems into the following terms:

1. A binding must be established between a process and an AGS;
2. There must be a means for a process P1 to determine the binding between another process P2 and its AGS.

When a user approaches Cronus to start a session a process (P1) is allocated(11). P1 cannot be bound to U (the user's principal identifier) until Cronus establishes the connection via password authentication. Before that happens, P1 is bound to a well-known principal, "NotLoggedIn", which has minimal authorization. One task of the login procedure is to change the binding of P1 from NotLoggedIn to U.

The binding between a principal identity and a process is established by the Authenticate As operation. The user engages in an authentication dialogue with Cronus, supplying a name and password which is checked against the UDB. If the authentication dialogue succeeds, the AGS for U is computed and a binding is established between P1 and U. A record of the binding

P1, U, AGS

is maintained by the process manager for the authenticated process, to be used throughout the process lifetime. The identity of the user has been established, completing problem 11.

(11). Cronus actually uses a more complex process structure to support a user session, as indicated in section 13.3. However, the following discussion is insensitive to these details, so we use this simple model in our explanation.

Throughout the course of U's session, P1 and other processes acting on behalf of U attempt actions which require authorization verification by the processes that perform the actions. This is problem I2. Consider a situation in which P1 has requested another process (S1) to perform some action (A), shown in Figure 9.

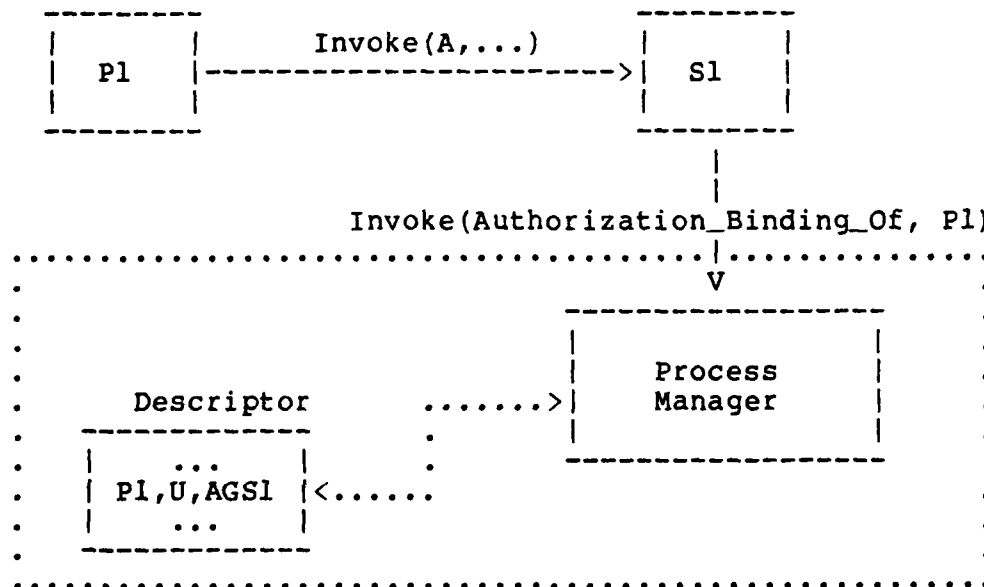


Figure 9 . Retrieving Access Control Data

In order to perform an access control check, S1 needs to determine the binding of P1. The identity of P1 is known to S1 because P1's UID was delivered along with the operation invocation that requests A. S1 can obtain the binding of P1 by invoking the Authorization_Binding_Of operation:

Authorization_Binding_Of(P1) -> U, AGS.

Authorization_Binding_Of causes a message to be sent from S1 to the manager for process P1, which returns the bindings for the process to S1.

The login sequence establishes a binding between user (U) and an "initial" process (P1). Bindings are established for other processes created during a user session through inheritance. During a user session, processes created by an authenticated process inherit both the principal identity and a specifiable subset of the current AGS of the initiating process. Object managers attain their principal identities and access group sets as part of the system initialization phase.

7.3 Authentication Manager

The Authentication Manager defines and maintains two types of abstract Cronus objects: CT_Principal and CT_Group. Like other system objects, the CT_Principal and CT_Group identifier objects have symbolic names for convenient human access. Principals are symbolically named from a private namespace maintained by the Authentication Service, which ensures their uniqueness across the entire system. Symbolic group identifiers can be placed anywhere in the Cronus catalog, at the convenience of the creating user.

Operations on objects of type CT_Principal and of type CT_Group are controlled by access control lists. By convention, any legitimate principal can create a new CT_Group object, but only administratively authorized principals can create a new principal. When the system is initialized, it contains at least one pre-defined principal, which is authorized to create other principals.

The next phase of Cronus will support a redundant Authentication Manager to ensure a survivable authentication capability.

7.4 Objects Related to Authorization

The object of type CT_Authentication_Data is the user data base consisting of the records for system users and for groups of principals which have been defined in the system.

The object of type CT_Principal is the permanent data base entry that Cronus maintains for each legitimate user. It is the repository for such user-specific data as default priority and other parameters associated with resource management; default modes of behavior (e.g. default working directory); and

authorization data. It is expected that new kinds of data will be added to the principal objects from time to time.

A CT_Principal object can be expected to contain the following data:

- o Principal unique-identifier (PID)
- o Symbolic name of principal
- o Access control list
- o Encrypted password
- o Direct group memberships
- o Direct group memberships to be expanded on Login
- o Range of priority service authorized
- o Default priority
- o Name of default initial subsystem
- o Name of home directory for the principal ... (other user-specific data)

The priority data will be used in resource management functions. The default subsystem is the program automatically invoked following login. A home directory is a directory assigned to the principal that serves as the initial current directory for catalog accesses; in particular, it contains additional user initialization data.

Groups (objects of type CT_Group) gather a number of identities for purposes of collectively granting them rights to objects and operations. Any user can create a new group, and place any other principal or group in it. This group can then be placed on an ACL. The access control list for the group object controls modification of the group definition.

A CT_Group object contains at least the following data:

- o GID for the group
- o Name of the group
- o GIDs of the groups of which the group is directly a member
- o IDs of principals (PIDs) and groups (GIDs) that are direct members of the group

There are a few special group identifiers. One of these represents the set of principal identifiers (that is, it is an all users group) without actually enumerating them anywhere.

This group identifier is automatically appended to every AGS computation. Another special group represents a "wheel" or "superuser" capability. Admission to this group is carefully controlled. A server with principal identifier which is a member of the supervisor group can be used to support a capability based manager for exporting rights from the cluster.

As a complement to All Users group, there is an entry which gives an identity access to all objects in the manager process domain. For example, a file manager identity might be given access to all file objects (present and future). This is useful in handling peer managers, which would be designated by a system principal identifier specific to the function they perform.

7.5 Operations on Authorization Related Objects

7.5.1 Operations on the Object of type CT_Authentication_Data

The following operations are used to create and locate the objects of type CT_Principal and CT_Group which comprise the authentication data base:

Create_Principal (...initial parameters...) -> new
principal_UID

When an object of type CT_principal is created, the creating principal is given all rights to the created object's operations. The new principal added to the access control list has Set_User_Parameters permission, as well as Show_System_Parameters and Show_User_Parameters permissions.

Lookup_Principal (principal symbolic name) -> Principal_UID

Convert the symbolic name representation of a principal into its unique identifier.

Create_group (... initial set of member UIDS) -> new group
UID

The creating principal is given all rights to the new group. Anyone can create new groups.

The following operation is used during login to establish the binding of the the user to the principal UID:

AD-A139 983

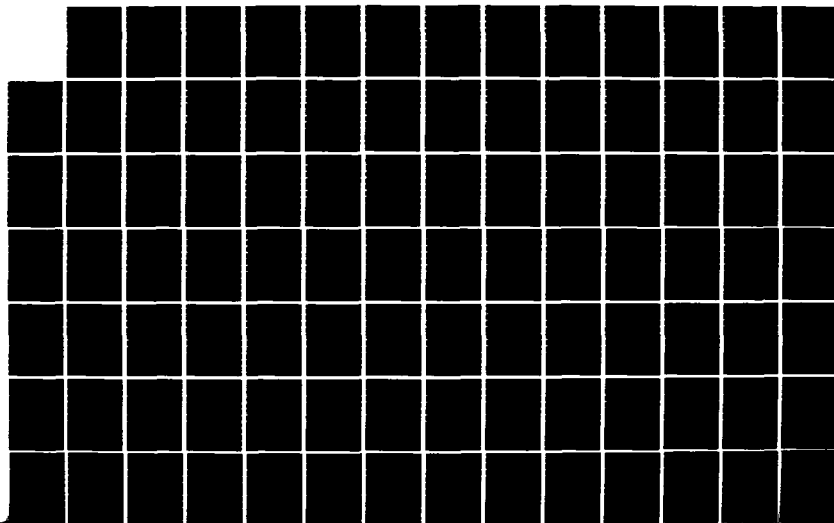
CRONUS A DISTRIBUTED OPERATING SYSTEM(U) BOLT BERANEK
AND NEWMAN INC CAMBRIDGE MA R SCHANTZ ET AL. DEC 83
BBN-5261 RADC-TR-83-255 F30602-81-C-0132

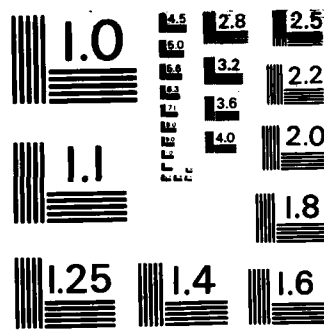
2/3

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Authenticate_As (user_name, encrypted password) ->
(principal_UID, list of user specific keys and values for
UDB data)

Execution of this operation is controlled by the password for the user entry. If successful, Authenticate_As computes the current AGS, based on the default expansion list for the principal. The principal_UID and the AGS for the invoking process are replaced by the new principal_UID and the current AGS. The Authenticate_As operation can be performed by any process at any time. The principal_UID is returned to the invoking process.

The following operations allow processes to control the identities applicable to an authenticated process. They effect only a single process, which may be either the invoking process or another process authenticated to the same principal.

Enable_Access_Group (list of group_UIDS, process_UID) ->
status

Expanded the group_UID and add the result to the AGS of the named process, provided the principal is already a member of the groups named.

Disable_Access_Group (list of group_UID, process_ID) ->
status

Remove the set of groups of the expanded group_UID (if they are present) from the AGS of the process named by process_ID.

7.5.2 Operations on Objects of type CT_Principal

The following operations maintain and interrogate the objects of type CT_Principal:

Delete_Principal (principal_UID) -> reply code

Permanently remove the object identified by principal_UID. This may be done only by system administrators placed on the access control list when the object was created.

Show_Group_Memberships (principal_UID) -> list of group_UIDs

Display the group UUIDs for all groups of which this principal is a direct member, noting which groups are in the default expansion list for the principal.

Add_to_Default_Group_Expansion_List (principal_UID, list of group_UID) -> reply code

Delete_from_Default_Group_Expansion_List (principal_UID list of group UUID) -> reply code

Change_Password (principal_UID, new password) -> reply code

The principal entry in the user data base contains user-specific fields. To limit the number of distinct operations and permissions needed to handle those fields, we specify two pairs of operations, one to Show and Set parameters which are under administrative control (e.g. allowable user priority range, disk quota,) and the other to Show and Set user controllable parameters (e.g. default priority, default home directory).

Show_System_Parameters (principal_UID) -> (parameter name, value pair list)

Show_User_Parameters (principal_UID) -> (parameter name, value list)

Set_System_Parameters (principal_UID) (parameter name, value list) -> reply code

Set_User_Parameters (principal_UID), (parameter name, value list) -> reply code

7.5.3 Operations on Objects of type CT_Group

The following operations are used to inspect and maintain the group identifier objects:

Delete_Group (group_UID) -> reply code

Add_to_Group (group_UID, list of new member _ UUIDs) -> reply code

Remove_from_Group (group_UID, list of member UIDS to remove)
-> reply code

Show_Group_Members (group_UID) -> (list of direct group
member UIDS)

Show_Group_Members_Expanded (group_UID) -> (list of direct
and indirect group member UIDS)

Show_Group_Membership (group_UID) -> (list of groups of which
this group is a direct member)

Show_Group_Membership_Expanded (group_UID) -> (list of groups
of which this group is a direct or indirect member)

7.5.4 Operations on Objects of Other Types

The following operations are the show and modify operation that apply to the access control lists for all the object types. These operations are themselves controlled by the access control list for the object being interrogated.

Show_Access_Control_List (object_UID) -> list of ACL entries

Add_to_Access_Control_List (object_UID, list of ACL entries)
-> reply code

Remove_from_Access_Control_List (object_UID, list of ACL
entries) -> reply code

When an object is created, the default access control list gives the creating principal all rights to it. Additional access control list entries can be entered by the creating agent using the access control list modification operations.

7.5.5 Operation of the Access Control Authorization Function

Cronus access control checks the current identity of the accessing agent against access control lists maintained by the service provider. A process is authenticated in a way which binds the process UID to a set of external identities defining the authorizations of the process. These identities, the AGS,

are available to any service-providing process. This section discusses the authorization function which is part of the service provider.

In general, the access control steps within a resource manager proceed as follows:

1. The request is parsed to determine the originating process UID and the operation/object requested. The process_UID is trusted because it is added to the message by the operation switch. Universal public privilege for the operation to all objects managed by the manager is first checked, to see if the specific access check is needed.
2. A manager-based cache of process/object authorization pairs for the process_UID is checked for a valid current entry.
3. If there is no corresponding cache entry, the accessing agent's AGS is obtained. This data is also cached but on a per-host basis by the AGS cache manager. If present on the host, this cache manager provides a high performance interface to the Authentication_Bindings_Of function. There is a broadcast-based protocol for alerting AGS cache managers to entries that should be purged. If an AGS cache manager does not run on a host, managers execute the Authentication_Bindings_Of operation directly, and the AGS is not cached.
4. The access control software computes a new process_UID/object authorization entry using the AGS and the access control list maintained with the protected object/operation. The process_UID authorization entry is then put in the manager cache.
5. The process UID object authorization is used to verify permission. If authorized, the operation is passed on to the operation code. If unauthorized, the request is rejected, and any cache entry is deleted.

The permission authorization function is accomplished by a set of routines and data structures that we call the "gatekeeper" because of its role as protector of the objects/operations. Gatekeeper functions can be invoked as part of the procedures for receipt of a message, or called directly from the host process.

Access control can be applied to operations on the object set supported by the receiving manager process, or on operations defined by the receiving service. There is a fixed maximum number of operations which can be access controlled by the gatekeeper software (currently 32) for any object. These operations are represented as positions in a bit vector associated with both the identity it authorizes (principal identifier or group identifier) and the object it controls.

7.6 Host Registration

The lack of physical security for various parts of the system presents problems for the access control subsystem. Since the network cable may be accessible to tampering, the network might be tapped. An outsider could then inject or inspect packets under an assumed network address. A workstation might pose as the site of a trusted manager. We can use administrative authorization to alleviate these problems.

Encryption of all local network traffic at the communication level is a form of authorization. It can remove the threat of tapping for either listening for or insertion of packets. Providing the host with the encryption/decryption key is administrative authorization to participate in the Cronus cluster. If a host can communicate at all, it can be considered an authorized host. Because encryption/decryption is isolated in the communication interface, it can be added transparently at any time. While communication encryption can be thought of as part of the Cronus design, it will not be part of the initial implementation.

Since workstations may be treated specially for some access control decisions, system configuration registry could be the source of such identification. In addition, the undesirability of tightly controlling responses to broadcast Locate operations, makes the registry useful in determining the authenticity of the respoondee. A configuration registry enumerates all of the authorized system hosts, and the system services (Cronus functions) which they have been authorized to run.

One secure way to make the registry service available is to support it on one (or more) well-known Cronus hosts (i.e. hosts at a well-known internet addresses, say host No. 1, ...). The configuration data can then be obtained with an Invoke On Host to

the well-known hosts using the logical name for the service(12). The cluster configuration service would support the following functions:

 Show_Configuration_Hosts (modifier or all) -> configuration
 data for all hosts or only for those indicated by
 modifier

 Set_Configuration_Hosts (modifier, data) -> reply code
 modifier indicates new configuration, add, delete, etc.

Standard access controls apply, with Show_Configuration_Hosts being universally allowed, while Set_Configuration_Hosts limited to a system administration group.

(12). Since this function is often used to determine the veracity of responses to the Locate operations, it can not safely use Locate to find out where configuration managers are running.

8 Cronus Primal File System

8.1 Cronus Primal Files

Cronus supports a number of different kinds of files, including:

- o Primal files.

The primal file is the most basic kind of Cronus file. Other kinds of Cronus files are implemented from primal files. A primal file is stored entirely within a single host, and is bound to the host.

- o Migratory files.

A migratory file can be moved from host to host. A migratory file is implemented by one or more primal files. Each primal file used to implement a migratory file contains all of the file data.

- o Dispersed files.

A dispersed file is implemented by one or more primal files. A dispersed file is one whose contents may be distributed over more than one host. Each of the primal files used to implement a dispersed file contains part of the contents.

The initial implementation supports only primal files, which are implemented upon underlying single-host file systems.

Primal files are Cronus objects. They have unique identifiers (UIDs), and may be given symbolic names. There is a Cronus object type CT_Primal_File.

Primal files cannot be moved from one host to another; the primal file system is partitioned among hosts that store primal files. The HostNumber component of the UID for a primal file always specifies the host on which the file is stored. A copy of a primal file can be created on another host, and the original can be deleted. The copy is a different primal file with a different UID; it just happens to contain the same data as the original file.

Like other Cronus objects, primal files are accessible to processes by means of the interprocess communication and operation switch (Section 6). There is a Primal File Manager

process on each host that stores part of the primal file system. A client process accesses a primal file by invoking an operation on the file, in which the UID for the file and the operation to be performed on the file are specified.

The Primal File Manager that maintains a primal file also defines a mapping between the UID for the primal file and the information required to manage the file. The collection of information necessary to manage a primal file is called its descriptor. The file descriptor includes:

- UID of the creator;
- Date and time of creation;
- Date and time of last write;
- Access control list (ACL) for the file;
- Information necessary to find the file data on the storage media;
- Current size of the file;
- Other information (to be specified as needed)
- ...

Most of the operations provided by conventional file systems (create, read, write, etc.) are implemented for Cronus primal files. The design is discussed in terms of the normal life cycle of a primal file which includes:

1. The file is created.
2. Data in the file may be read or written by a client.
3. Information in the file descriptor may be read or written by a client.
4. The right to access the file may be granted to or revoked from other users.
5. The file may be deleted.

File creation involves: the generation of a UID; the creation and initialization of a descriptor for the file; the binding of the UID and the file descriptor in the Primal File UID Table. Until data is written into the file, the file is empty. When a primal file is created by a Primal File Manager, it is created on that manager's host.

There is an issue regarding whether it should be necessary to open a primal file before reading or writing file data. One reason for "open" and "close" is to provide for reader-writer synchronization; another is optimization of read/write operations. The disadvantage is that open/close add complexity

to the Primal File Manager because it must maintain state information for open files and deal with the problem of files opened which are never explicitly closed (e.g., because the client's host has crashed). Furthermore, if we require open and close, additional operations must be invoked on the file even when the read or write is for a small amount of data.

The Primal File Manager supports access to files without open and provides an open/close facility for clients that need it. A read or write without open is called a "free read" or a "free write". The client may then chose whether the additional overhead of opening and closing the file is worthwhile. For example, if we wish to write a simple log message when a process is initiated, we would probably chose the free write. If, on the other hand, we were copying a file, we would probably chose to open the files, incurring the overhead of initiation once, and gaining further system support for synchronization and data integrity. A client process may read or write data in a primal file (subject to authorization considerations) without opening it, unless another process has opened the file in such a way that free reads and writes are forbidden.

Free reads and writes are synchronized in the sense that multiple reads and writes are serializable. This means that the File Manager will, in effect, perform each read or write operation in its entirety before performing another operation.

When a file is opened, two parameters specify the access state requested. One specifies either Read or ReadWrite access. The second specifies the type of reader-writer synchronization desired. There are two types of synchronization supported: "frozen" which permits either N readers or a single writer; and "thawed" which permits any number of simultaneous writers and readers. When a file is opened with "thawed" access, readers of the file see updates made by writers of the file.

Thus, the access states defined for a file are:

```
free;
frozen read open;
frozen readwrite open;
thawed open;
read in progress;
write in progress.
```

A file may be opened so long as the access state requested does not conflict with the current access state of the file.

Table 2 defines the compatibility of the access states with one another, and with read and write operations invoked by a client without previously opening the file. An OK for an (OPERATION, ACCESS STATE) entry in the table means that a client process can perform the operation on a file when the file is in the corresponding access state; a NO entry means that the operation will fail when the file is in the corresponding state; a DELAY operation means that the operation will be delayed until the operation in progress (and any others that may be queued) are completed.

OPERATION	ACCESS STATE					
	free	frozen read	frozen readwrite	thawed	read in progress	write in progress
frozen read open	OK	OK	NO	NO	OK	DELAY
frozen readwrite open	OK	NO	NO	NO	DELAY	DELAY
thawed open	OK	NO	NO	OK	DELAY	DELAY
free read	OK	OK	NO	OK	OK	DELAY
free write	OK	NO	NO	OK	DELAY	DELAY

Table 2. Access State Compatibility

The data in a primal file as a sequence of bytes, numbered from 0 to N. The read operation specifies the first byte to be read and the number of bytes to be read. The write operation specifies the byte position of the first byte to be written and N bytes of data to be written.

In order to support data system recovery, data that is written to a file that has been opened for (ReadWrite, Frozen) access does not become part of the permanent file data until the file is closed. It is possible to close a file opened for

(ReadWrite, Frozen) access in a way that aborts writes made to the file while it was open.

A file is open to a process. The Primal File Manager provides an operation which returns a list of the UIDs for the processes, if any, that have a file open. Another operation returns a list of the UIDs for the files, if any, that a process has open.

When a process is destroyed with files open, the files are closed and any writes to (ReadWrite, Frozen) open files are aborted. The normal close operation may only be invoked by the process that opened the file. An alternate close operation can be used by other processes to close a file during cleanup.

A client can read the descriptor of a primal file. Some of the information in the file descriptor is changed as a side effect of operations on the file. For example, when a file is written, the date and time of last write is changed. There is other information that the client may wish to change explicitly.

Access to a primal file is controlled by its access control list (ACL). Access to a primal file may be granted to other users by adding entries to the ACL. Similarly, access to a file may be revoked from a user by removing the corresponding entry from the ACL.

This document assumes that only Delete will be supported, but it is relatively straightforward to modify the specification of Cronus primal files to accommodate a Delete, Undelete, and Expunge model of file removal.

8.1.1 Executable Files

Executable programs will be stored as files of type CT_Executable_File which is a subtype of primal File. There will be many different kinds of hosts in Cronus, and generally an executable program file which can run on one host type will not be able to run on another. In addition to the normal descriptive information, files of this type have information that specifies where they can be run. The additional information maintained for an executable file would include:

- o The type of processor required to execute the program stored in the file.

- o The run-time environment required by the program including the local operating system and necessary peripheral devices.

8.2 Crash Recovery Properties

If a primal file operation is invoked, the Primal File Manager normally acknowledges the operation, indicating the disposition of the operation (e.g., success, failure, and reason) and, depending upon the operation, to return any data requested.

The Primal File Manager does not acknowledge write requests until the data has been written to non-volatile storage. A client process can be sure that the data has been written when the acknowledgement is received, even if the Primal File Manager or its host should crash shortly afterward.

Primal File write operations are atomic with respect to host crashes. That is, if the Primal File Manager host should crash during a write operation, after the host and Primal File Manager have been restarted and the Primal File Manager has performed its recovery procedures, the write operation will have either occurred in its entirety or no part of it will have occurred. If the crash occurs after the data has been safely written but before the acknowledgement has been sent, the acknowledgement will never be generated.

This atomicity property is true for the Close-and-RetainWrites operation. That is, either none or all of the writes made while the file was open will have been performed.

8.3 Operations for Objects of type CT_Primal_File

The following operations are supported for primal files:

The Open and Close operations provide an atomic transaction capability for a single primal file. At some later point, we may define explicit BeginTransaction, EndTransaction, and AddToTransaction operations which could be used to provide a

capability for transactions that involve more than a single primal file.

8.3.1 Operations on Object of Type CT-Primal_File_System

The following operation is defined on the object of type CT_Primal_File_system:

Status(HostID) -> StatusInformation

Create(HostID CL_Primal_File) -> UID

Create a primal file and return the UID for the new file. The file is empty until data is written into it. The ACL for the file given the creator has every right that is defined for a primal file.

Delete(UID) -> ReplyCode

Deletes the file specified by UID.

Open(UID, TypeOfAccess, TypeOfSynchronization) -> ReplyCode

where TypeOfAccess is:

Read, or
ReadWrite

and TypeOfSynchronization specifies the reader-writer synchronization for the file and may be:

Frozen, which means N readers, or 1 writer is permitted, or

Thawed, multiple writers and readers permitted.

Close(UID, Mode)

where Mode is

RetainWrites or AbortWrites.

RetainWrites causes the file data to be updated. AbortWrites causes the file data to remain as it was prior to being opened. (This mode is only meaningful

when the file open was for ReadWrite and Frozen).

Read(UID, Position, Amount) -> Data

Position specifies a starting byte position, Amount specifies the number of bytes to be read, and Data is the file data returned.

Write(UID, Position, AmountToWrite, Data) ->
AmountWritten

Position specifies a starting byte position (the value -1 is used to indicate the current end of the file); Data is the data to be written AmountToWrite is the number of bytes to be written; and AmountWritten is the number of bytes actually written.

Truncate(UID, Length) -> Reply Code

Truncate the file to Length, discarding all data beyond that point.

Append(UID), AmountToWrite, Data -> amount written

Append data to the current end of the file. This operation is equivalent to Write with a position of -1, but permission may be granted separately.

ReadDescriptor(UID) -> file descriptor data structure

Returns the file descriptor for the file.

WriteDescriptor(UID, WriteSpec) -> ReplyCode

WriteSpec specifies the changes to be made to the descriptor for the file.

ReadACL(UID) -> ACL

Returns the access control list for the specified file.

AddToACL(UID, ACL_Entry)

Adds the ACL_Entry to the access control list for the file.

`RemoveFromACL(UID, ACL_Entry)`

Removes the `ACL_Entry` from the access control list for the file.

`FilesOpenBy(HostID, ProcessUID) -> List`

Returns a list of the primal files managed by the Primal File Manager on `HostID` that are currently open by the specified process. The list of element is of the form: (`PrimalFileUID`, `TypeOfAccess`, `TypeOfSynchronization`).

`OpenStatusOf(UID) -> List`

Returns a list of the processes which currently have the file open. The list element is of the form: (`ProcessUID`, `TypeOfAccess`, `TypeOfSynchronization`).

`CloseProcessOpenFile(FileUID, ProcessUID, Mode) -> Reply Code`

Close the file, retaining or aborting writes as specified by `Mode` `HostID`, if it is currently open by the specified process.

`CloseAllProcessOpenFiles(HostID, ProcessUID, Mode) -> Reply Code`

Close all files open by the process that are managed by the File Manager on `HostId`.

The Primal File Manager returns information about the status of the primal file it manages, such as the amount of free space, the amount of space used by existing files, the number of files it manages, the number of files it manages, the number of files currently opened, etc.

This information will be useful to system operations personnel as well as to clients who might use it when deciding where to create primal files.

9 Symbolic Naming

9.1 The Cronus Symbolic Name Space

9.1.1 General Syntactic Conventions

Cronus has a global symbolic name space with the following properties:

1. Cronus symbolic names are location independent.
 - a. A name for an object is independent of its host.
 - b. A name that refers to an object can be used regardless of the location from which it is used.
2. Cronus symbolic names are uniform.

Common syntactic conventions apply to names for different types of objects.

The symbolic name space is constructed upon a hierarchically structured tree. The tree contains nodes and directed labeled arcs. There is a distinguished node called the "root". Each node has exactly one arc pointing to it, and can be reached by traversing exactly one path of arcs from the root node. Nodes in the tree represent Cronus objects which have symbolic names. A link facility transforms the name space into a network, so a node may be reached by more than one path.

Non-terminal nodes (those from which arcs may originate) are called directories. Each labeled arc corresponds to a catalog entry. The label for an arc is called an "entry name".

The complete name of a node, which is the symbolic name for the object, is formed by concatenating the labels on the arcs traversed on the path from the root node to the node in question, separated with the character ":". In other words, the syntax for a complete name is:

$$: \{ x : \}^* y$$

where "x" and "y" are arc labels, the "{", "}" brackets indicate optional presence, the ":" is a punctuation mark to separate name components, and "{ s }*" means zero or more occurrences of s.

It is also possible to name nodes relative to a directory.

Such a relative or partial name is formed by concatenating the labels on the arcs traversed on the path from the directory in question to the node. The syntax for a partial name is:

{ x : } * y

There are conventional names for the current ("connected" or "working") directory, its parent, and the user's initial directory.

9.1.2 Types of Objects Cataloged

The most common types of cataloged objects are the various kinds of files, but any other object may be cataloged. Some conventions will be adopted; for example, there will be a :dev directory which contains the symbolic names for the devices on the system. These conventions are not enforced by the system, and any object may be entered into any directory (assuming appropriate authorizations) at the convenience of the user.

There are certain special object types which are used in support of the catalog itself, including:

- o Directories

A directory object (type CT_Directory) is the collection of catalog entries which correspond to the arcs that originate from a non-terminal node in the name hierarchy tree.

- o Links

The catalog entry for a link (type CT_Symbolic_Link) identifies another point in the symbolic name space called the link target. These objects are stored in the catalog itself. Links are cataloged as terminal nodes in the name hierarchy tree. Links are handled specially within the Lookup operation.

- o External linkages

An external linkage (type CT_External_Linkage) is an object which implements access to another name space. External linkages are cataloged as terminal nodes in the name hierarchy tree. External linkages permit users to refer to non-Cronus objects directly from the Cronus name space. For

example, an external linkage might be used to give a file directory on a Cronus application host a Cronus symbolic name.

For some object types it is useful to be able to think of a collection of the objects as a sequence of "versions" or "revisions" of the same logical object. The Cronus Catalog implements a version feature for certain types of objects; for example, versioning will be supported for files, but it will not be supported for directories.

For types for which versioning is supported, the catalog entry operation will permit the same name to be entered into a directory more than once. The first time a name is entered, the result will be version 1 of the object. Subsequent entries of the same entry name will result in successively higher versions of the object. All of the catalog operations which take a name parameter will allow the specification of a version number as well.

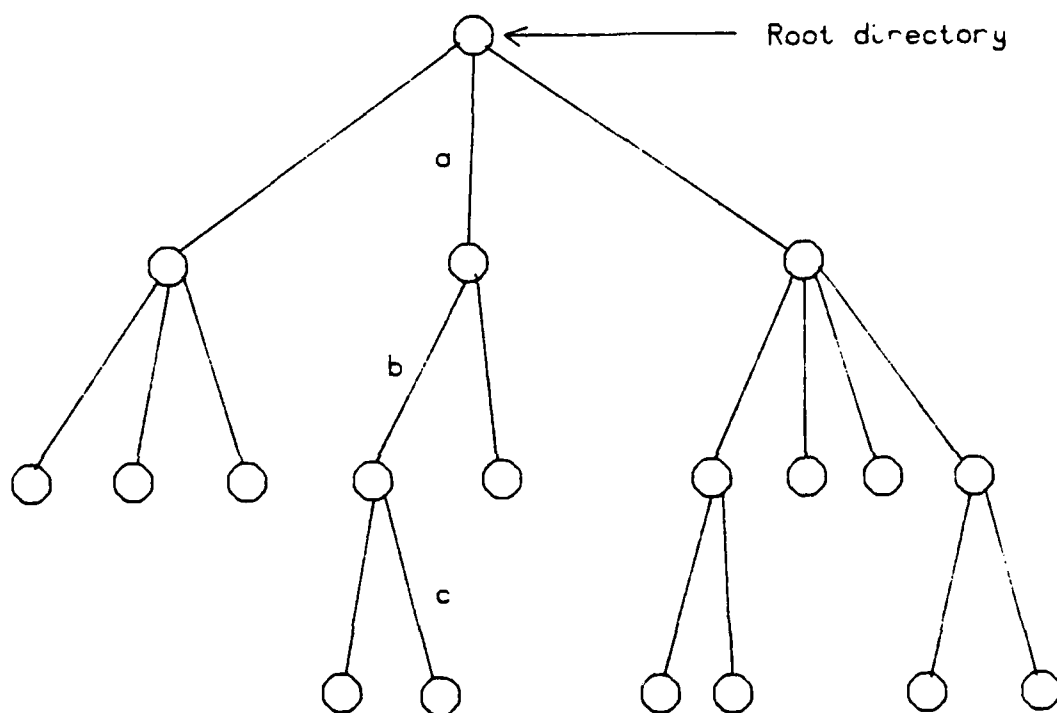
The catalog managers provide routines that can scan through the catalog and return catalog entries for names that match a specified pattern.

9.1.3 Directories and Links

The catalog entry operation can be used to establish a symbolic name for a Cronus object of any type except a directory, symbolic link, or external linkage object. These objects must be created by special operations because they are inserted in the catalog when they are created (since other objects need not be named, the creation of the object and naming of the object are separated). In a sense, these objects are special in that they must have a symbolic name in addition to a UID.

Figure 10 shows a relatively simple symbolic name tree and Figure 11 shows part of the underlying directory structure that corresponds to the part of the tree that contains the name :a:b:c.

When a lookup operation is invoked, the catalog manager interprets a complete Cronus symbolic name by starting at the root directory. The UID of the root directory is well-known.



Catalog Hierarchy
Figure 10

Implementation of Cronus Catalog

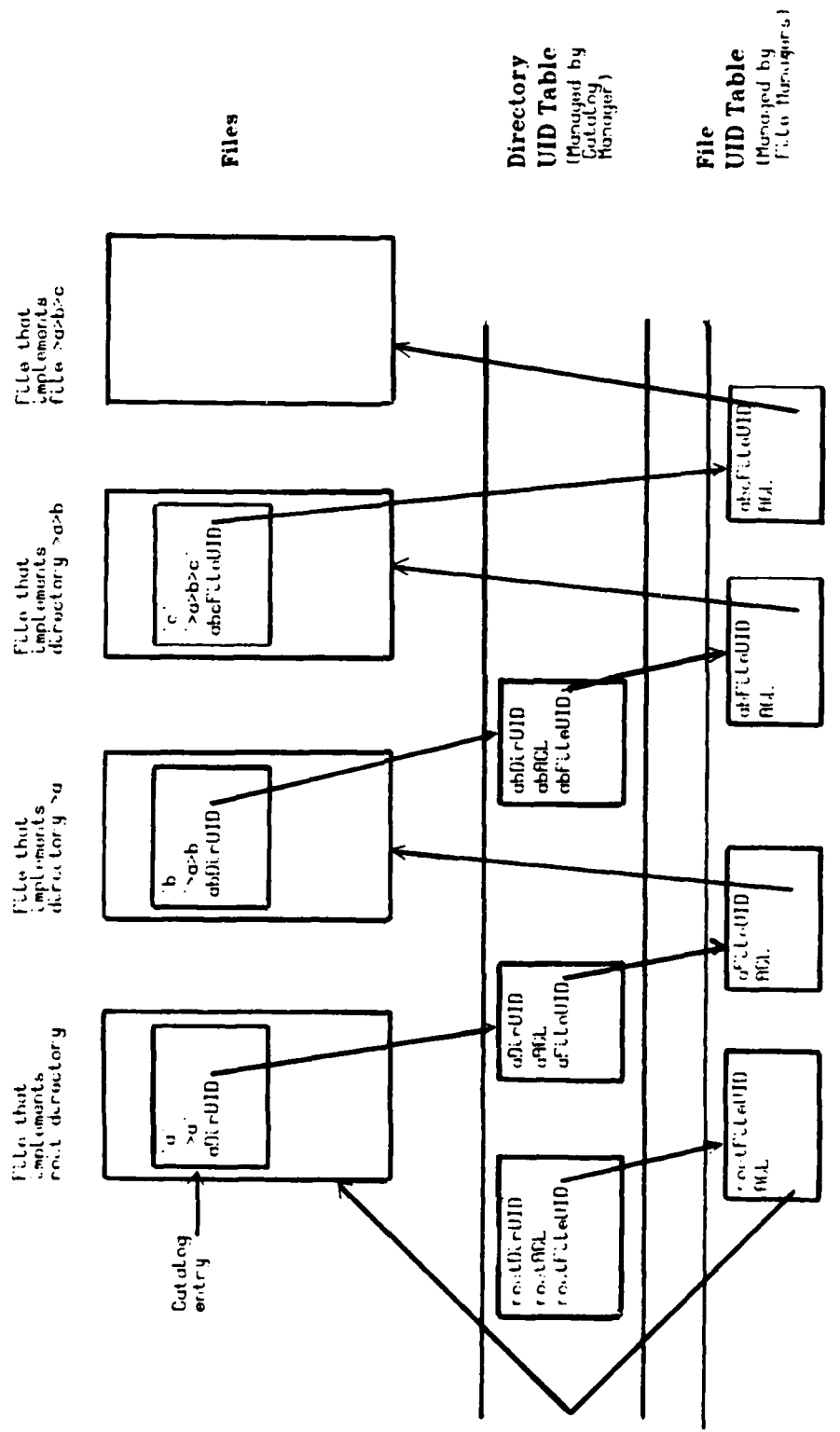


Figure 11

The catalog manager processes a name component by searching the current directory for a matching catalog entry. If it finds a matching entry and there are no more name components, the lookup is complete and it returns the catalog entry. If it finds a matching entry and if there are more name components to interpret, the entry must be for a directory, symbolic link, or external linkage, or else the lookup ends in failure. If the entry is a directory, the catalog manager continues the lookup by obtaining the UID for the directory from the entry and then using it to interpret the next component. Interpretation of a partial symbolic name is handled in the same fashion, differing only in where the lookup starts. For a partial name, the catalog manager starts its search at the starting directory parameter of the lookup operation.

Symbolic links encountered during lookup are handled specially. When a link is encountered, a new name is formed by substituting the link target, which is a complete Cronus symbolic name held in the catalog entry, for the portion of the symbolic name evaluated so far. The lookup operation then resumes by interpreting this new name. Links can be thought of as macros which are expanded during the lookup operation.

A parameter of the lookup operation controls whether links are to be expanded. If the parameter specifies that links are to be expanded, the substitution of link targets during the lookup operation occurs. If the parameter is set to prevent links from being expanded, the lookup operation terminates when a link is encountered. In this case, the lookup operation will be considered successful if the name has been completely evaluated. Otherwise, it will be considered a failure.

9.2 Objects Related to the Catalog

9.2.1 Objects of Type CT_Catalog_Entry

Each catalog entry is a Cronus object. A catalog entry will contain the following information:

- UID for the object;
- Complete symbolic name for the object;
- UID for creator of entry (PrincipalUID); and
- Type-dependent information.

Type-dependent information for objects of type CT_Directory,

CT_Symbolic_Link, and CT_External_Linkage is discussed below. For objects that are not part of the Cronus catalog, everything that can be known about an object is maintained by (or can be obtained from) the manager for the object. The issue is, what part of this information, if any, should be replicated in their Cronus catalog entries? This question is answered on a case-by-case for each object type.

The disadvantage of maintaining information in the catalog is that the information becomes obsolete as the objects undergo modification. Maintenance of such information is more difficult when the object has more than one symbolic name, and hence, more than one catalog entry.

Performance and reliability are improved if we maintain information in the catalog about objects. The performance advantages occur because the overhead of interacting with the object manager can sometimes be avoided. The reliability advantages occur when the object manager for the object is inaccessible, but the catalog entry for the object is accessible.

What, then, is the nature of the coupling between the information about an object in its catalog entry and the information held by the object manager? The simplest approach couples the information very loosely and places responsibility on the client process. Since this places the burden on all clients, the information is likely to be unreliable. The Cronus catalog software provides a tight coupling. When a name is established for an object, the catalog manager will send an ObjectCataloged message to the object manager. The object manager then sets an ObjectCataloged flag in its descriptor for the object, and sends back a message containing the information that should be stored in catalog entry.

As an example, the catalog entry for a primal file might contain type-dependent information, such as:

- UID of the file creator (a PrincipalUID);
- Date and time of creation;
- Date and time of last write; and
- Current size of the file.

When an object is modified whose ObjectCataloged flag has been set, its object manager will send the information necessary to update catalog entries for the object to the catalog manager. The information about an object held by its manager is the truth

and any information held in catalog entries for it, except its symbolic name and UID, is advisory and maintained as a convenience. The system is structured so that its correct operation does not depend upon information found in the catalog.

9.2.2 Objects of Type CT_Directory

For directories, no type-dependent information, except possibly the host that stores the directory, would be maintained in the catalog entry. All other information about the directory will be maintained with the directory object itself.

9.2.3 Objects of Type CT_Symbolic_Link

For a symbolic link, the type-dependent information, which completely specifies the link, a link is the complete symbolic name for the link target.

- UID;
- Complete symbolic name for the link;
- UID for creator of entry (PrincipalUID); and
- Complete symbolic name for the link target.

9.2.4 Objects of Type CT_External_Linkage

For an external linkage, the type-dependent information completely specifies the external linkage. It includes a Cronus interpretable designator for locating the other name space and a symbolic name that is interpretable in that other name space. The details of the method for designating other name spaces and for interacting with them are incomplete. A catalog entry for an external linkage will include:

- UID;
- Complete (Cronus) symbolic name for the external linkage;

UID for creator of entry (PrincipalUID)
Cronus interpretable designator for the other
name space; and
Symbolic name interpretable in the other
name space.

9.3 Catalog Operations

9.3.1 Objects of Type CT_Catalog_Entry

The following operations are defined for the Cronus symbolic catalog:

Enter(DirUID, EntryName, ObjectUID) -> CatEntUID

Establishes a symbolic name for an object. A check is made to determine whether EntryName is already in use in the specified directory. If EntryName is not in use, a catalog entry is created. If EntryName is in use, and the type of the object cataloged under EntryName is the same as the type for ObjectUID, and versioning is supported for that type, then a new entry for a new version of EntryName is created; otherwise, the operation will fail. This operation is not defined for objects of type CT_Directory, CT_Symbolic_Link, and CT_External_Linkage.

Remove (DirUID, CatEntUID) -> ReplyCode

Remove Cat_EntUID from DirUID. The corresponding name for the object is also removed from the symbolic name space. This operation is not defined for objects of type CT_Directory.

Lookup (StartDirUID, Name, FollowLinks)
-> DirUID, CatEntUID, CatEntContents

Find the Name in the catalog. FollowLinks controls whether links are to be expanded during the lookup. If Name begins with ":", it is a complete symbolic name and the lookup begins in the root directory. Otherwise, Name is treated as a partial name. In

this case, StartDirUID is the start point for the lookup. The DirUID returned is the UID of the directory that contains the catalog entry. CatEntContents is the data structure for the catalog entry. It includes the object UID, the complete symbolic name, and possibly other type-dependent information. If any links were expanded during the lookup, the symbolic name in the CatEntContents will not be the same as the Name parameter.

ReadEntry(DirUID, CatEntUID) -> CatEntContents

Returns the contents of the specified catalog entry.

ChangeEntry(DirUID, CatEntUID, NewContents) -> Reply Code

Modifies the type dependent information in a catalog entry.

InitScan(StartDirUID, PatternSpec)
-> ScanState, DirUID, CatEntUID,
CatEntContents

Initializes a catalog scan, and returns the DirUID, CatEntUID, and contents for first catalog entry, if any, that matches PatternSpec. If it begins with a ":", the pattern is for complete names and the StartDirUID parameter is ignored. Otherwise, PatternSpec specifies partial names and StartDirUID is required. ScanState represents the current state of the scan and must be supplied on subsequent interactions with catalog to obtain additional catalog entries matching PatternSpec. ScanState can be tested to determine when the scan has ended.

ScanDirectory (ScanState)
-> ScanState, DirUID,
CatEntUID, CatEntContents

Perform the next step of a catalog scan and returns the next catalog entry, if any, that matches the pattern. The ScanState specifies the current position of the scan, and the returned value of ScanState parameter indicates the position after the step has been taken.

LookupWild(DirUID, PatternSpec)

-> list of
(DirUID, CatEntUID, CatEntContent)
tuples

This operation initiates and performs a catalog scan.

EntriesOf(ObjectUID)

-> list of (DirUID, CatEntUID) pairs

Returns the UIDs of all catalog entries for the specified object. The result may be zero, one, or more catalog entry UIDs. This operation does not return pairs which are the result of links.

ChangeObjectEntries(ObjectUID, NewContents) -> Reply Code

Update every catalog entry for the specified object. It will be used by object managers to keep information held in catalog entries for object current.

9.3.2 Objects of Type CT_Directory

The following special operations are defined for objects of type CT_Directory:

CreateDir(OldDirUID, EntryName [, HostID]) -> DirUID,
CatEntUID

Creates a new directory by entering it. A catalog entry for into the OldDirUID under the name EntryName. The optional HostID specifies the Cronus catalog host that is to store the new directory. The HostID parameter is examined only if the dispersal cut (see Section 9.4) is below OldDirUID. If it is not supplied and OldDirUID is above the cut, the new directory is created above the dispersal cut. If the HostID parameter is supplied, then the new directory is created below the cut and is stored on the specified host. Versions are not supported for directories.

DeleteDir(ContainingDirUID, DirUID) ->

Delete a directory, which succeeds only if the directory is empty.

9.3.3 Objects of Type CT_Symbolic_Link

The following special operation is defined for objects of type CT_Symbolic_Link:

```
EnterLink(DirUID, EntryName, TargetName)
        -> CatEntUID
```

Establishes a link in DirUID with name and target TargetName, which must be a complete symbolic name. Versions are not supported for links.

9.3.4 Objects of Type CT_External_Linkage

The following special operation is defined for objects of type CT_External_Linkage:

```
EnterExternalLinkage(DirUID, EntryName,
                    ExternalNameSpaceSpec,
                    ExternalName)
        -> CatEntUID
```

Establishes a new external linkage in DirUID. ExternalNameSpaceSpec specifies the external name space. ExternalName specifies the target for the external linkage, and is a name that is interpretable within the external name space. Versions are not supported for ExternalLinkages.

9.3.5 Access Control for Catalog Operations

All of the catalog operations are operations on one or more directories. There are three rights defined for access control purposes:

```
ReadDirectory,
WriteDirectory, and
ModifyACL.
```

ReadDirectory rights are needed in Lookup for each of the directories required to interpret the Name. ReadDirectory rights are needed in ReadEntry for the directory that contains CatEntUID; in EntriesOf for the directories that contain any

CatEntUIDs that are returned; in InitScan for the start directory; in LookupWild for all directories encountered; and in ScanDirectory for the directory specified in the ScanState.

WriteDirectory rights are needed in Enter for DirUID; in Remove for the directory that contains CatEntUID; in ChangeEntry for the directory that contains the entry to be changed; in CreateDir for OldDirUID; in ChangeObjectEntries for each directory that contains a catalog entry that is changed; and in DeleteDir for the directory that contains the directory being deleted.

The Table 3 summarizes the access rights required for the various operations.

	Read Directory	Write Directory
Enter		x
EnterLink		x
EnterExternalLinkage		x
Remove		x
Lookup	x	
LookupWild	x	
InitScan	x	
ScanDirectory	x	
ReadEntry	x	
ChangeEntry		x
CreateDir		x
DeleteDir		x
EntriesOf	x	
ChangeObjectEntries		x

Table 3. Access Rights Required for Catalog Operations

9.4 Catalog Implementation

9.4.1 Introduction

The following implementation issues are discussed below:

1. the use of Cronus data storage resources to implement the catalog data base;
2. the distribution of the catalog data base among Cronus hosts; and,
3. the manner in which client processes interact with the catalog manager which implement the catalog functions.

9.4.2 Implementation of the Catalog Hierarchy

Directories are implemented by files. The catalog manager maintains a UID table for the objects it manages. Since the principal objects implemented by the catalog manager are directories, this table is called the Directory UID Table. The Directory UID Table maps the UIDs for directories and their object descriptors.

A directory contains zero or more catalog entries. The catalog entry for a (inferior) directory contains the UID of that directory. To access a directory given its UID, the catalog manager uses the Directory UID Table to obtain the object descriptor for the directory, and then uses the file UID in the descriptor to access the file that holds the directory.

The catalog manager also maintains a Cataloged Object Table which implements an object-UID-to-catalog-entry mapping, which has an entry for each Cronus object that has a symbolic name. The entry contains the UID of the cataloged object and a list of (DirUID, CatEntUID) pairs for each catalog entry for the object. The Cataloged Object Table is updated as part of the Enter, Remove, CreateDir and DeleteDir operations, and it is used to implement the EntriesOf and ChangeObjectEntries operations.

9.4.3 Distribution of the Catalog

9.4.3.1 Principles Affecting Distribution

Among the considerations influencing catalog distribution are:

1. The catalog should not be stored at only one site.

This is a reliability consideration.

The catalog should be distributed, and it should probably be replicated in some fashion.

2. The entire catalog should not be stored at any single site.

This is a scalability consideration.

3. It should always be possible to access an object when the site that stores the object is accessible.

This is a reliability consideration.

Access to objects through the UID name space has this property since the information required to access an object, given its UID, is maintained used by object managers. Access to objects through the symbolic name space should also exhibit it.

The catalog entry for an object (or a copy of the entry) should be stored at the same site as the object. In addition, there should be enough information at the object site to control access to the object.

4. There is little utility in maintaining a catalog entry for an object in a more reliable fashion than the object itself.

This is a common sense consideration.

It is not necessary to replicate catalog entries for objects beyond that required by (3).

The next two subsections discuss considerations (2) and (4) in more detail. The discussion includes elements of the

implementation of the reliable system as well as the primal system, because these may impose constraints on the primal system design.

9.4.3.2 Dispersal Of The Catalog

This section examines the requirement that the catalog not be stored at a single site. The line of reasoning followed is essentially that that lead to the design of the Elan hierarchy [BBN 3796].

Directories are the basic unit of distribution for the Cronus catalog. Directories are implemented by Cronus primal file so a directory is stored entirely within a single host. The lookup operation follows the components of a symbolic name through a number of different directories, one for each component in the name (assuming it does not encounter a symbolic link). Unless there is a further restriction on the dispersal of the catalog, each directory could be at a different site from the previous one.

It is desirable to limit the number of sites that must be visited in a lookup operation. Two useful restrictions are to:

1. Require that the catalog structure for entire subtrees below a certain cut (the "dispersal cut") through the catalog tree be stored within a single site. We call a subtree that is rooted at the dispersal cut a "dispersal subtree".
2. Require that the catalog structure above the dispersal cut be stored within a single site. We call the structure above the dispersal cut the "root portion" of the hierarchy.

Restriction 1 ensures that lookup operations within a subtree that is below the dispersal cut can be confined to a single site. Restriction 2 ensures that the task of determining the site that stores a particular dispersal subtree can be confined to the site that stores the root portion of the hierarchy. As a result, lookup operations require at most two catalog sites.

It is useful to add a third property to the dispersal of the catalog:

3. The root portion of the catalog hierarchy should be replicated. Furthermore, a good way to replicate it is to maintain it at each site that maintains a part of the catalog (i.e. a dispersal subtree). The reasons for doing this are:

To distribute the load resulting from lookup operations among several sites.

To allow some lookup operations to be confined to a single site.

To increase the availability of the root portion of the hierarchy.

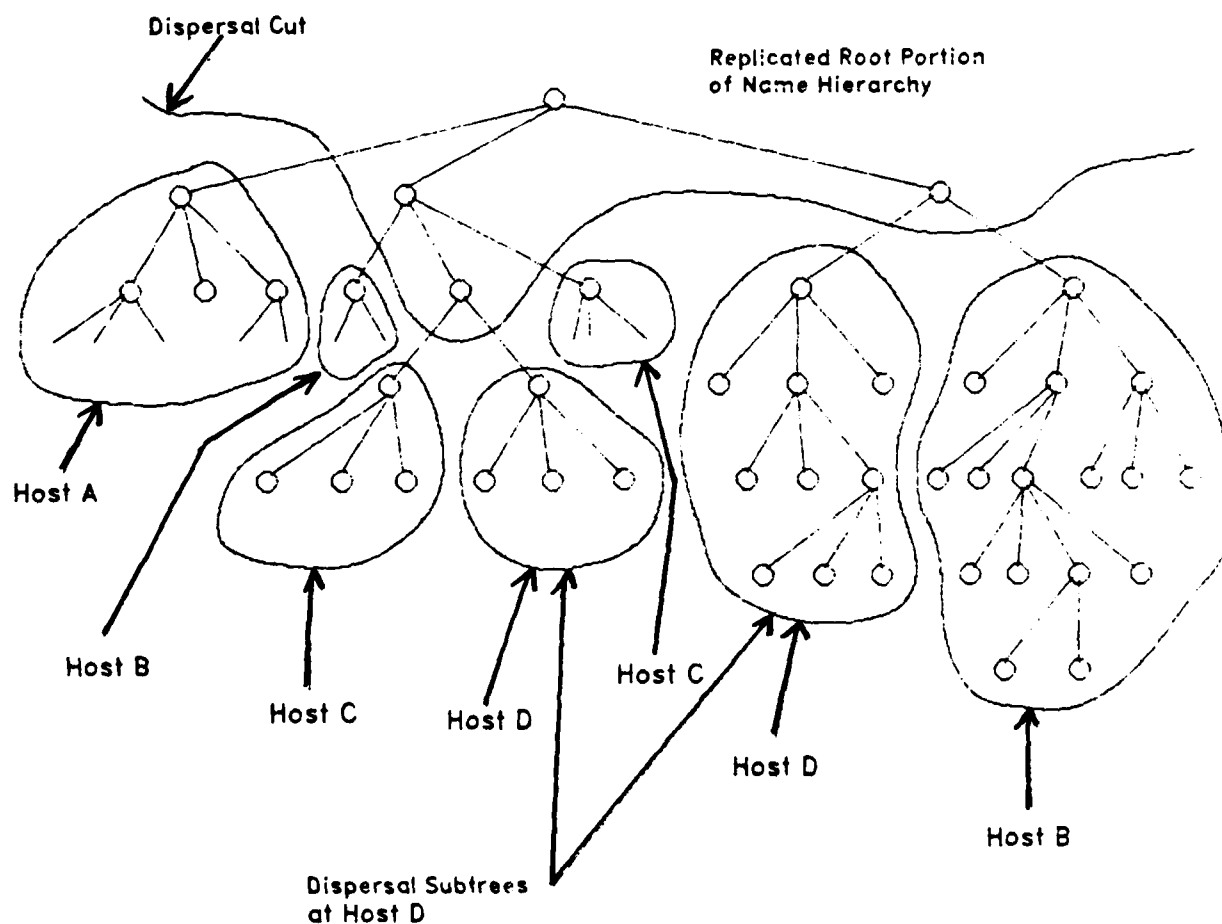
Figure 12 illustrates how a simple name hierarchy might be dispersed among several hosts according to these three restrictions.

For this to be practical, it must be possible to maintain the copies of the root portion in a consistent fashion among the same set of hosts that store parts of the catalog. It has been observed that the root changes very slowly, because few users are authorized to make changes, and because changes generally occur as the result of the addition or deletion of a user or project. This means that the maintenance mechanism need not be powerful enough to handle the general multiple copy update problem.

9.4.3.3 Dispersal of the Cataloged Object Table

The Cataloged Object Table supports the EntriesOf and ChangeObjectEntries operations. The EntriesOf operation returns any entries in the Cataloged Object Table for an object specified. The ChangeObjectEntries operation uses this information entry for the specified object to find the catalog entries that need to be modified, and then it modifies them.

When a name is established for an object, an addition is made to the Cataloged Object Table. If the object already had a symbolic name, an addition is made to its existing entry. When a name is removed, the corresponding information is removed from the Cataloged Object Table. The entry for an object is removed when its last symbolic name is removed.



Dispersal of the Catalog
Figure 12

Logically, the Cataloged Object Table can be viewed as a single table which contains an entry for each object that has a symbolic name. However, like the catalog itself, the Cataloged Object Table will be implemented in a distributed fashion. The following are three approaches to distributing the Cataloged Object Table.

1. Total Replication

The Cataloged Object Table can be replicated in its entirety at every catalog site, so it is accessible whenever any catalog site is. This simplifies the EntriesOf and ChangeObjectEntries operations. Maintaining full copies of the table is relatively expensive both in terms of storage space, and difficult to do in a consistent fashion.

2. Fragmentation Among Catalog Sites

Each site that stores part of the catalog can also store the corresponding part of the Cataloged Object Table. It is then relatively easy to maintain the individual fragments of the table. The only catalog activity that modifies a site's fragment is a change to the part of the catalog managed at the site. The disadvantage of this approach is ChangeObjectEntries operations are more complex; there may be entries for an object in fragments at several sites.

3. Fragmentation Among Object Sites

Each site that stores an object can maintain the Cataloged Object Table entry, if any, for that object. Its use by EntriesOf and ChangeObjectEntries is relatively straightforward since the entire entry for an object is stored at the site that manages the object. The disadvantage is that, in general, changes occur as the result of operations performed by catalog managers that are remote from the entry. For example, whenever a catalog entry is added to or removed from a directory by a catalog manager a corresponding change must be made to a Cataloged Object Table entry which will, in general, be remote from the catalog manager. Consequently, cooperation between catalog managers and software at the object hosts is required to maintain the Cataloged Object Table fragments.

This approach meshes well with the scheme for providing

secondary symbolic access paths to objects described below. Activity that requires modification to the Cataloged Object Table at a site also requires modification to the collection of catalog entry copies at that site. The Cataloged Object Table and the collection of the catalog entry copies could be implemented by a single data base, structured so that it can be searched in two ways: by object UID to obtain the corresponding Cataloged Object Table entry; and by symbolic name to obtain the corresponding catalog entry copy.

Our inclination at present is to avoid the fully replicated approach (1), and to continue considering the two fragmented approaches (2 and 3).

9.4.3.4 Replication of Catalog Information

For the purposes of the current implementation, we can defer consideration of the problems associated with cataloguing multiple copy objects.

The primary consideration for replicating catalog information is one of reliability. The objective is to ensure that Cronus objects with symbolic names are accessible symbolically whenever the sites that manage the objects are.

There seem to be two approaches to providing symbolic access to objects when the Cronus catalog is inaccessible.

1. Replicate the catalog sufficiently to ensure that it is available with the degree of reliability that is desired. This would involve maintaining multiple copies of directories.
2. Replicate the catalog information required to access a particular object (i.e., the information in its catalog entry) to the degree desired and store it at the host that stores the object.

We rule out the first approach for two reasons:

1. Directories below the dispersal cut will change

relatively frequently, making it difficult to maintain multiple copies of them in a consistent fashion,

2. In later versions of Cronus, a directory may hold catalog entries for single copy objects and for multiple copy objects that are replicated differing amounts, making it unclear how many copies of the directory should be maintained.

In the second approach, we maintain a secondary symbolic access path to objects rather than replicate the catalog structure itself. The primary symbolic access path to an object can be represented schematically as:

```

      Cronus
Name  --> catalog  --> UID  --> UID  --> Object
          entry      Table

```

The secondary symbolic access path would be supported at the host managing the object by a copy of the Cronus catalog entry for the symbolic name. If the object has more than one symbolic name, a copy of each catalog entry will be stored at the object's host(13). The secondary path can be represented schematically as:

```

      Distributed
      copies of
Name  --> Cronus  --> UID  --> UID  --> Object
          catalog      Table
          entries

```

That is, there will be a collection of Cronus catalog entries at each host for those objects that have symbolic names that require access to directories on other hosts. The catalog manager software will maintain the consistency between these distributed catalog entry copies and the Cronus catalog.

Figure 13 illustrates how the cataloging information will be maintained. The circular nodes represent objects that are stored at the same host as their entry in the catalog hierarchy and the

(13). If all of the directories required to find a particular symbolic name for an object are located on the same host as the object, there is no need to maintain an additional copy of its catalog entry at the host to support a secondary access path.

square nodes are used to represent catalog entries for objects that are stored remotely from their entries.

Under normal conditions, the lookup operation uses the symbolic catalog. When not all of the directories are available, the secondary symbolic access path is used. The lookup will succeed whenever the object itself can be reached, since if the object has a symbolic name, a copy of the catalog entry object will be stored at the site that manages the object.

Lookup by means of the primary path is much more efficient since it is directed, whereas lookup by means of the secondary path is undirected. There is no a priori knowledge of the host or hosts that need to be consulted to perform a lookup by the secondary path. Furthermore, because the collection of catalog entry copies does not hold complete information about the full structure of the naming hierarchy, it will be difficult to organize the copies into a data structure that can be searched as efficiently as the hierarchical catalog database.

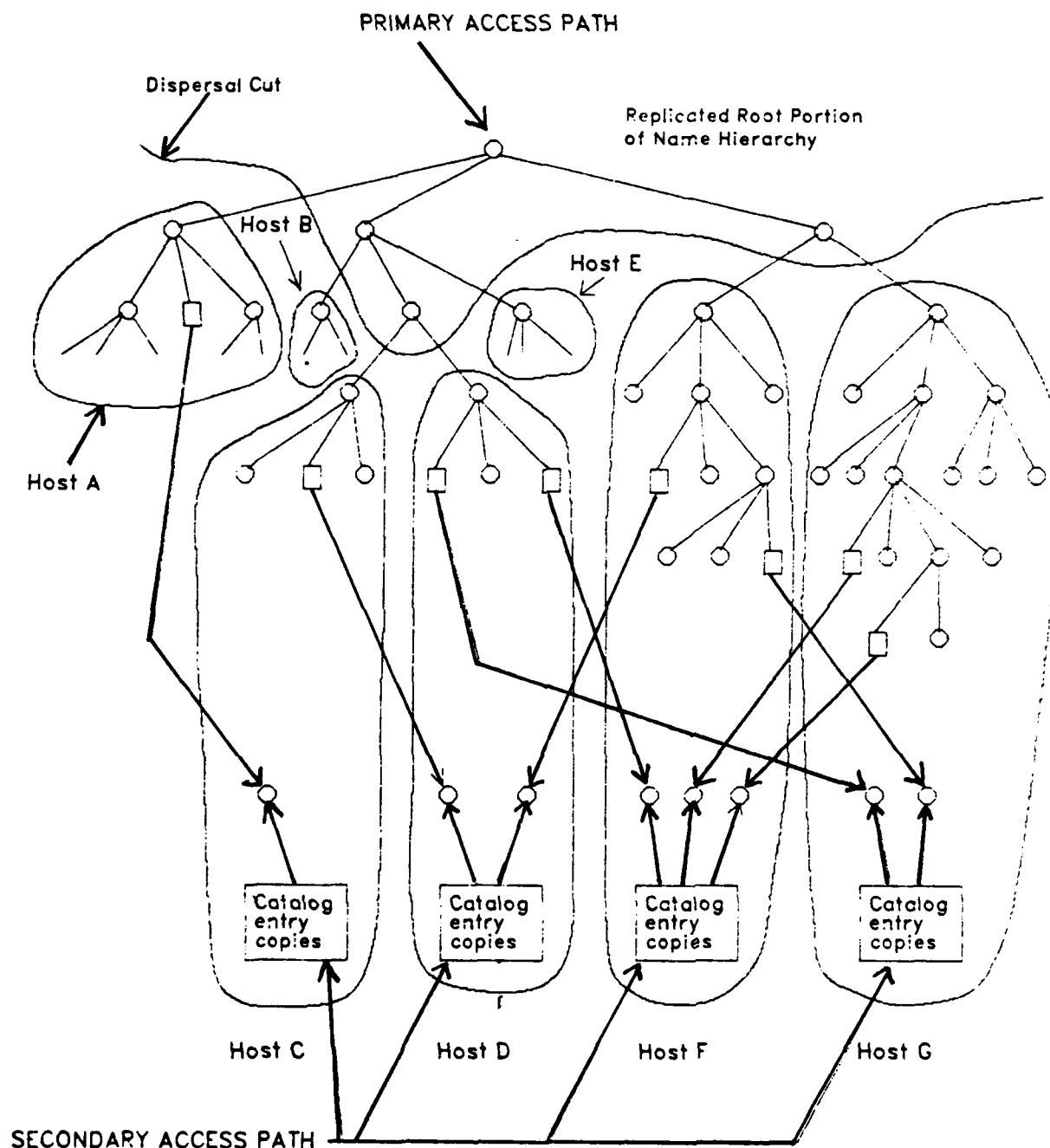
9.4.4 Cronus Catalog Managers

There is a catalog manager process at each host that maintains part of the catalog. It is the object manager for objects of CT_Catalog_Entry, CT_Directory, CT_Symbolic_Link, and CT_External_Linkage.

The catalog managers communicate with client processes by means of the standard Cronus IPC facility. Since the catalog hierarchy is distributed among Cronus hosts, different managers will have direct access to different parts of the catalog. Some catalog operations can be accomplished by a single catalog manager and some require the cooperation of two or more catalog managers.

For example, the Remove(DirUID, catEntUID) operation would normally be sent to the manager for directory DirUID, and only that manager is required. The lookup operation may require catalog managers on two hosts if the manager to which it is sent does not contain the subtree required to interpret the entire symbolic name. Finally, the ChangeObjectEntries operation may require the participation of every manager.

A client process will not, in general, know which catalog manager is the best one to perform a given operation. For this



Secondary Symbolic Access Path
Figure 13

reason, a client can initiate a catalog operation with any catalog manager. If the manager selected can perform the operation requested by itself, it will. If not, it will interact with other managers as necessary to perform the operation.

10 Input/Output

The addition of I/O devices the Cronus DOS must be considered at three levels: the Cronus user, internal Cronus, and constituent OS levels. In this section, I/O integration is examined at each level, and the interfaces to other components of the same level and to other levels is described. Line printer and tape drive devices are used as examples.

Devices are Cronus objects, and there are object manager for each device type. Device names are entered in the symbolic catalog.

Interactions with a device within Cronus are with the manager for the device, which hides the device driver of the constituent operating system from the client. The generalized location-independent framework is also independent of particular devices and of device-specific functions.

Each device type has a Cronus logical name (i.e., a well-known UID with type CT_Type_Name), and each instance of a device has a specific UID. There is a single manager on a host for each device type having a device instance on that host. The manager will select an available device and initiate the operation. Some device types may associate a separate Cronus process with each instance of a device; in this case, the generic request to the manager will be forwarded to one of the device processes.

Each device has a device manager which maps generic input/output operations into the corresponding calls into the I/O Process Support Library. There four generic operations:

- o Open (DeviceUID, ProcessUID) -> DeviceTaskUID
- o Close (DeviceTaskUID) -> Reply
- o Read (DeviceTaskUID NumberOfBytes) -> DataBytesRead
- o Write (DeviceTaskUID, Data, NumberOfBytes) -> BytesWritten

The device manager receives an Open request to initiate a device task for a process. The device manager determines who the principal is and applies the standard access control mechanisms. On the successful completion of device initialization, it keys a task state record to DeviceTaskUID, and replies to the requestor to proceed. Subsequent requests for input or output must be accompanied by DeviceTaskUID. When the manager receives a Close command, from the user process, the device is released, the task is terminated, and all state information associated with it is purged. The task state information will also be purged if the device manager learns that the user process died without closing

the device.

Device managers are normally passive entities and must be directed by explicit requests from user processes. Although this is the default behavior, it is desirable in some cases for the device manager to play a more active role. Some device managers, therefore will be able to initiate an activity.

Consider, for example, how a printer spooler might work. A user, desiring that a file be printed, executes the "print" file command which puts a copy of the file in the spooler directory. The spooler process selects the file from the spooler directory, and sends an open request to the primal file manager. After a successful open reply, the spooler sends an open request to the line printer manager. When a successful line printer reply is received, the spooler prints the file by requesting data from the primal file manager and sending to the printer process. After the file is completely printed, close operations are performed on both the file and the printer, and the spooled file is deleted.

Assume that the spooler, file, and printer manager are not all on the same host. In this case it is inefficient to require that each data block go through the spooler. Instead, the spooler gives the line printer device manager the file object UID with directions to initiate a "copy" on that object. When printing is complete, the line printer device manager notifies the spooler.

The four generic operations listed above are required for device managers. There may be any number of additional device-specific operations supported by a particular device manager. For example, in addition to the read and write operations, a tape drive manager must support the tape positioning functions, tape read/write density, and so on. Any process that is permitted direct access to the tape drive device manager is allowed to invoke the device-specific functions. For instance, an archive process would open the tape drive for reading, and then, on direction of a user, retrieve files or even directory trees from an archive tape.

Symbolic names for devices are in the Cronus Symbolic Catalog, in the directory :dev. Assuming there are several line printers in Cronus, the names ":dev:lpt3" or ":dev:gce2_lpt1" then refer to specific instances of line printers, and are bound to the appropriate device UIDs. The symbolic name for the default line printer is ":dev:lpt". Bound to this name is the type name UID for line printer, which can be used to find an

instance of a line printer.

The constituent operating systems (COSs) are responsible for handling devices at the most basic level. For each Cronus device, there must be device driver. The details of the implementation of this driver COS- and device-dependent. Once the device driver is established, there is a COS dependent access path to the device. This access path provides for both device control and data transfer.

The COS dependence is hidden by the Process Support Library functions. The implementation of this library varies from one COS type to another, but it presents a uniform interface to the Cronus device process.

11 User Interface

The user interface for Cronus consists of several parts. System access requires a mechanism for enabling the user to interact with Cronus. This is normally provided by connecting a terminal to Cronus. The user interface also provides a command interface, which allows the user to control the session.

The objective is to provide users with flexible, convenient access paths to the system. Cronus will support a number of different types of access points including:

1. Terminal access computers (TACs): A Cronus terminal multiplexer connected directly to the DOS local area network. TACs are implemented in dedicated GCEs.
2. The Internet: The Cronus local network is connected to the Internet by means of a gateway computer. Users outside the cluster may access Cronus through the standard terminal handling protocol (Telnet) which operates upon a lower level, reliable transport protocol (TCP).
3. Mainframe hosts: Cronus mainframe computers are likely to have terminal ports, which enable access to Cronus through Telnet, like other hosts on the Internet.
4. Dedicated workstation computers: A workstation is a computer that is, at any given time, dedicated to a single user. Workstation hosts have sufficient processing and storage resources to support non-trivial application programs, such as editors and compilers, and to operate autonomously for long periods of time(14).

User interaction is supported by software that runs on one or more computers. This software includes two principal modules. One module is responsible for handling the user's terminal. Since this module will often run at the user's access point, we call it the "access point agent". The other module interacts with the user at a higher level to provide access to Cronus resources in response to user commands. We call this module the "session agent". It is useful to think of the access point agent and the session agent as processes.

(14). The Primal system will not support workstations.

For a user whose access point is a TAC, the access point agent runs on the TAC and the session agent runs on a shared host. Users who access Cronus through the Internet are allocated user agents that run on shared hosts, and their access point agents may run either on the (non-Cronus) host used to access the DOS or on a host within the DOS cluster.

The standard user interface software will be written to operate with CRT terminals that have cursor positioning capabilities. More capable terminal devices (e.g., graphics displays) can emulate the standard terminal device to obtain a compatible interface. In addition, a means will exist for users with other less capable terminal devices (e.g., printing terminals) to access the system. In the latter case, some sacrifice in the quality, uniformity, and power of the user interface is unavoidable.

The purpose of a user interface to Cronus is to provide human users with uniform, convenient access to the functions and services. User requests should be similar regardless of the particular Cronus components that implement them. For example, the way a user instructs Cronus to run a program should be the same (except for the name of the program) regardless of where within the cluster the program will execute. A user should not have to pay undue attention to the mechanics of establishing access. For example, to run an interactive program, a user should not have to explicitly establish a communication path with the host. Similarly, to delete a file a user should not have to explicitly establish communication with a file manager.

To be uniform and convenient does not mean that a user interface must make the network or the distribution of the system invisible to users. Often users will want the distribution to be transparent, and the user interface should provide transparency. There will also be situations where it will be important for the distribution to be visible to users and for users to exert control over how the system deals with distribution. For example, system operators and maintainers will need to deal directly with the system's distributed nature. Furthermore, ordinary users may want to control where programs run or files are stored.

A variety of different user interface programs can be constructed to manipulate the Cronus functions previously described. Cronus has been designed such that almost all of the user interface is provided by application level programs, which permit the coexistence of many different user interfaces and an evolving approach toward developing them.

The development of user interface functions will be based on the following principles:

- o Since most requests cannot be performed directly by the user interface it acts on the user's behalf to initiate activity by other modules.
- o The user interface enables a user to initiate and control multiple simultaneous tasks. In particular, a user may have several application programs executing concurrently.
- o The command interpreter may be selected at login time. Users with strong preferences for different styles of interaction can be accommodated simply by running different user interaction modules.
- o The user interface functions developed for the ADM DOS will be designed to operate best with a high speed CRT display terminal, with cursor positioning capability. It will make use of multiple windows on the display surface to display user interactions with the separate activities being controlled by the user. In addition, windows will be used to display system status and user help information.

12 Monitoring and Control

12.1 System Capabilities

The monitoring and control system (MCS) for Cronus includes monitoring and control of hosts and of the Cronus functions on these hosts, of the network substrate, and of gateways. The monitoring and control station provides the functionality of an operator's console for the Cronus Distributed Operating System. The MCS treats Cronus as an integrated system, decomposed by function rather than by host. Where practical, it also monitors and controls Constituent Operation System (COS) functions from the same station, but such functions are limited by our desire to modify the COSs as little as possible. The discussion in this section includes elements of the Reliable System as well as of the Primal System. These additions are included to assure that the Primal System design does not interfere with future extensions.

Cronus is restarted from the Monitoring and Control System. For some hosts, the MCS will invoke functions already on the hosts; in other cases (for example, GCEs which have no disks), the MCS will download the host to start Cronus.

Network monitoring and control of a local area cable-based network such as the Ethernet is relatively simple. It includes a detection and reporting of changes in host availability; monitoring and controlling traffic levels on the cable. Cable utilization and the traffic level of each host is measured. Priority or allowable traffic density may be set for each host. Transmissions from a host may be stopped altogether.

12.2 System Model for Monitoring and Control

Cronus consists of a set of services⁽¹⁵⁾ and low-level system support entities, including the Cronus IPC mechanism. The MCS is a set of processes on a Cronus host; its functions can be executed from anywhere in the cluster.

(15) A Cronus service is a process which performs Cronus operations in response to requests from other Cronus processes. All object managers, for instance, are services.

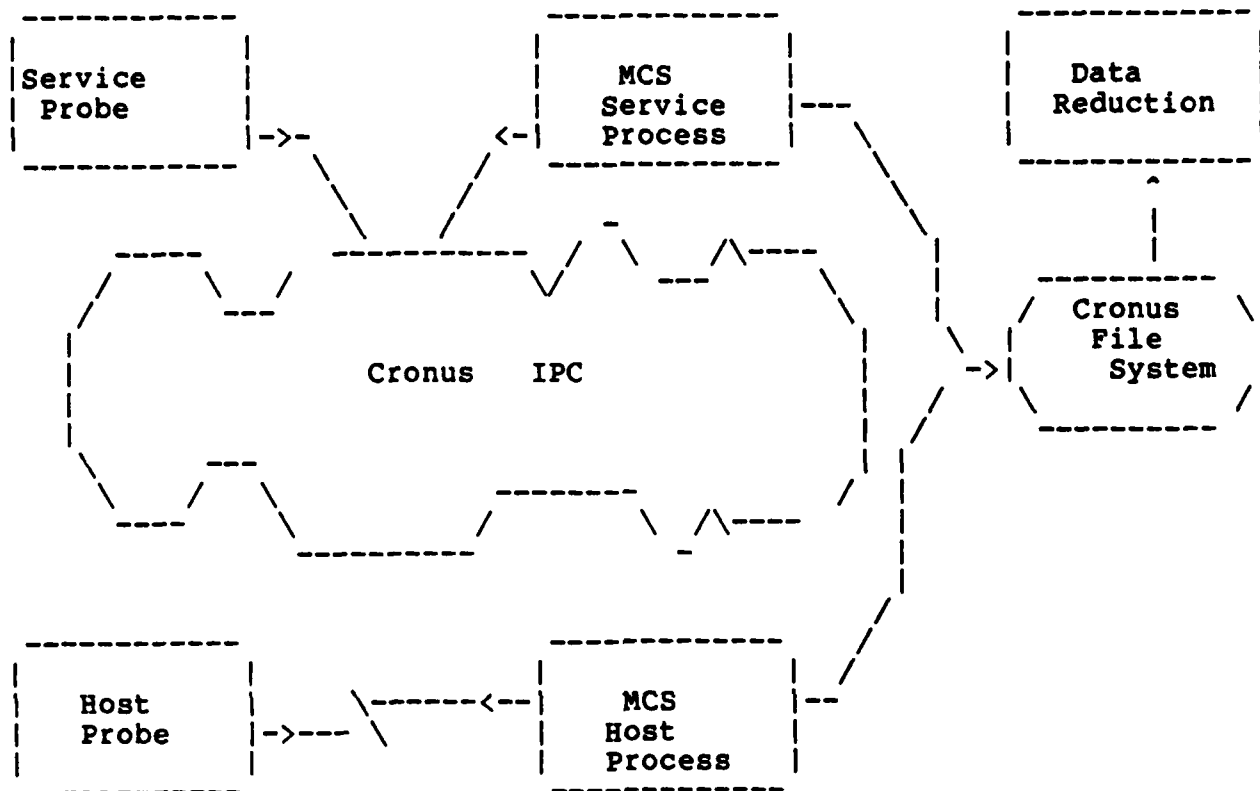


Figure 14 . Structure of the MCS

The MCS monitors both the support layer and the services. The set of services is extensible, and the MCS is designed to accommodate new services.

The MCS is based on a functional decomposition rather than on a site-based decomposition of the system. For example, one service monitor monitors all file system managers while another monitors authentication managers. The MCS will be aware of distinctions between sites and to distinguish them in its reports.

12.3 Structure of the MCS

The MCS runs as one or more Cronus processes. The MCS station is not bound to any particular site, although certain information gathering functions are most conveniently performed at one location. It uses the Cronus file system, in which it will store data, and the Cronus IPC facility. The MCS will be divided into two parts. The first part is the interactive section, which does on-line data collection, display, and control of Cronus. It obtains status information from host and service probes, and incorporates it into its own data base. The second part performs data reduction and generates reports.

The interactive section of the MCS consists of a very low-level module and a higher level module (see figure 1). The majority of the MCS resides in the high-level module, a Cronus service which communicates with its probes through the Cronus interprocess communication facility. The low-level module uses only the lowest level of network protocol (User Datagram Protocol). This primitive lower level can be relied upon when little of Cronus is functioning. This portion will be implemented first. It provides the functions required to bootstrap Cronus, to examine and alter memory on Cronus hosts, and to do simple monitoring of the Cronus network.

There are two types of reports to the MCS: polled messages and traps. Polled messages are reports in response to a request from the MCS. Traps are reports from probes which are unsolicited. They normally represent unexpected or unusual events.

The MCS uses polled messages as the primary data gathering technique. The polling request provides a mechanism which will quickly recognize when a host or service disappears.

Traps are employed for reports about specific events, which may require real-time response, or which are unanticipated. For instance, the crash of a service would be reported as a trap, so that service restoration or reconfiguration could be instituted immediately. A host coming up would similarly be reported by a trap message, because of the timeliness of the information and because a new host on the network might not get any unsolicited polls(16).

(16). Polling for hosts which are known to Cronus but currently down would continue at a low rate, however, so that a lost trap for such a host coming up would not be fatal.

The MCS contains a trap logging service. Trap reports are generated by both host and service probes. Trap messages include a service type and priority in their header, so that display routines can easily determine which traps require immediate display in a high-priority window, and so that the operator can easily select all traps in a priority range from a given service class (e.g. file system). The trap logger could be extended to permit automatic operations in response to traps, so that a "service crashed" trap report could be used to force a restart of the service from the MCS.

The display processes normally directs critical reports to the system operator, with each process controlling one or more text streams. A text stream may be directed to a display terminal window, a hardcopy output device, a file, or several different places. The operator terminal should support a multi-window display, which will enable the operator to monitor a variety of aspects of system operation simultaneously, with one window usually reserved for critical reports. Other windows will be created to present data as requested. For instance, an operator might choose a process in one window which presents the general status for all hosts in the network, and another window to present the load status for a particular host of interest.

When the sophisticated window package is not available, a simpler interface would enable the operator to monitor one window at a time; the difference would be invisible to the MCS since each window would look to it like an independent display.

The data reduction facilities of Cronus can reside wherever convenient, and will be regarded as background tasks. The integrity of the system does not depend on their availability, but their reports should prove useful to the tuning and management of the network.

The data reduction section will take advantage of the fact that the files generated by the interactive section are available globally as part of the Cronus file system.

12.4 Host Probes, Service Probes, and Network Monitoring

A host probe is a primitive entity which every Cronus host must provide to report status to the MCS. A host probe must at least report the presence of the host and its internet address at the time the host operationally enters Cronus, and must respond to AREYOUTHERE messages broadcast from the MCS. The host probe

is the distributed part of the low-level section of the monitoring and control system. A host probe will often offer further information in its report: host type, probe reports available, current MCS reports, Cronus services, level of integration, etc.

Service probes are monitoring entities in all Cronus services. Services to be monitored will include object managers, terminal concentrators, and user authenticators. Service probes reflect a functional rather than site-based decomposition of Cronus. Data from related service probes on different hosts are combined in the MCS, in order to present a more understandable picture of the service. The MCS specifies what types of data should be collected and reported through poll responses and through traps.

A service probe is located within the service. Unlike host probes, they may require a certain level of Cronus functions, since the loss of service monitoring and control does not compromise our ability to restart the system. Service probes use the full range of Cronus services, especially the Cronus IPC facility.

Some messages, including control messages and high-priority monitoring, will run with a priority above that of the service. Most monitoring, however, will run with a priority below that of the service itself.

The service probes for the Cronus file system reports the loading on the local portion of the file system, the number of requests for various classes of services, etc. It may also include the ability to trace all activities on particular files (using traps) as a debugging aid.

The process manager probe reports machine process loading, both for Cronus and non-Cronus processes, and optionally supports tracing services for activities on Cronus processes. The probe will report certain classes of exceptional events on processes, and will provide services, invokable from the MCS, for invoking and killing processes, and for tracing process activity on a per-process basis.

Gateway monitoring would normally fall into the category of service monitoring; however, the gateway already reports status in response to polling by a host. We will use this capability to obtain gateway and internet status. Since we do not wish to do development in this area, we will to restrict ourselves to the available capabilities.

The MCS will not monitor the cable network traffic directly. Rather, it will gather reports from hosts on the traffic sent, traffic received, and the collision rate at each node.

12.5 Loading and Debugging Support

The control function has the capability for restarting Cronus on the hosts of the network. It may do this in one of two ways. In some cases (e.g. GCE), this includes transmitting the code directly to the host to be loaded. In other cases, the computer's own loading sequence is invoked, using its private secondary storage. In no event should the downloading procedure require the assistance of a third machine. Some machines may detect some of their own failures and restart themselves.

A distributed, heterogeneous system such as Cronus poses special problems for debugging tools. The goal is to have a sophisticated debugger which runs on one host and debugs on another. We would like to have a single debugging system be capable of debugging computers of differing architectures. Moreover, we would like the debugger to be able to debug at source language level to provide for efficient development. Currently, the leading candidate for developing such a tool is XMD, which is adapted from the multi-window editor PEN. XMD does not currently debug code in high-level languages, but can be extended in this direction, since it does not depend on the structure of the debugged code, relying instead on symbol table entries to provide it with information about the target code. XMD may soon be extended to debug C source code as part of the effort of another project at BBN.

12.6 Cronus Initialization

The initialization of Cronus is performed from the Monitoring and Control Station. In initializing the system, the MCS will have no certain knowledge of what hosts are available. The first step is to poll for the available hosts, and then to initialize each host which responds.

The initialization of Cronus proceeds as follows(17): (See the scenario in Section 13.)

1. The MCS broadcasts AREYOUTHERE onto the network.
2. Each host has a routine in its COS that listens for AREYOUTHERE and responds with HEREIAM and the parameters (a) name, (b) internet address, (c) boot class, (d) boot file name, and any other required information. The name is printable. The boot class indicates the method used to initialize the host. Class 1 hosts accept a BOOTYOURSELF command and initialize local Cronus software upon its receipt. Class 2 hosts require a BOOTLOAD command, which is followed by a boot file (item d) which passed to the to the host with the code to load. Class 3 hosts require a host-specific loading protocol, which is executed on the MCS from the boot file. (There are no plans to implement Class 3 hosts in the ADM.)
3. When the MCS receives a HEREIAM message, it enters the addresses of the host in a host monitor table, with a notation that the host is not up. It then sends a BOOTYOURSELF message if it is a class 1 host, or a BOOTLOAD followed by the required file if it is a Class 2 host.
4. When a host has completed Cronus initialization, it sends a message BOOTDONE to the MCS. Alternatively, it may send the message BOOTFAIL, possible with parameters indicating reason (e.g. "missing file block 5"). The MCS may then retry the boot, if appropriate.
5. After the host is initialized, the MCS will communicate with it using the Cronus IPC mechanism. It will normally obtain a list of available services and will then ask it to start up the services it supports.

The initialization procedure requires a small amount of code resident in each processor in order to respond to the MCS messages. This code will fit in ROM on machines which do not have secondary storage.

(17): These messages do not use the full Cronus IPC mechanism in the first four steps of the procedure, since the operation switch and primal process manager are not in place on the host being initialized. Instead, they will be implemented as VLN messages.

12.7 Siting the Monitoring and Control System

Should the MCS be located on the GCE or on an application host? Using a GCE is desirable because it can be specially configured to support the MCS; it is intended to be the dedicated processor; it provides controlled, predictable performance with dedicated, low cost hardware; and it is expected to be redundantly available. Since UNIX hosts may not be available redundantly, we would less often have back-up service if use it on a UNIX application host for the MCS. On the other hand, building the MCS on an application host has several advantages: the UNIX host provides a much richer development environment; have already been written for UNIX, so that less program development would be necessary; we can take advantage of the set of available UNIX utilities.

For the near term, we will build the tools on UNIX. We will be careful to code the routines in a portable manner, so we can easily move them to a GCE environment. This provides us with the benefit of using UNIX in the short term, while keeping the eventual goal of relying on redundant GCE's for Cronus services.

12.8 Phased Implementation

Implementation of the monitoring and control station will occur in phases, both in terms of functionality, and in terms of reliability and performance. The functionality will be increased both as the reporting capabilities of the probes increases, and as the need for data analysis grows.

Initially, the MCS will exist on a single host, without strong reliability or performance goals. We will first build the host monitoring section of the MCS, and simple host probes in order to be able to start and restart Cronus hosts and services, and to record the status (up/down) of hosts. As services are written, we will add service probes, and extend the MCS to monitor them. This initial system will utilize the UNIX file system until the Cronus primal file system is available, and will then convert to the use of Cronus files. Later the MCS will reside on a GCE and will use standard Cronus files.

13 Scenarios of Operation

13.1 Basic User Commands and Functions

This section presents examples of the use of Cronus functions and of the integration of structural units. Scenarios are presented for typical system and application tasks. The intent is to suggest the interactions through the flow of control and shared data. The scenarios also suggest how the primitive functions might be combined to support operations required of modern operating systems. The first few sections are narrative, and the later ones provide pseudo-code examples. Details of syntax and calling sequences in these examples are not those of the actual implementation.

Many of the user commands and functions of Cronus fall into the following categories:

- o Session initiation and termination: Login, Logout, Attach, etc.
- o User and system data base status and maintenance: Display and edit user records, access control lists, show logged on users, etc.
- o File manipulation and file/directory maintenance: name lookup, read, write, directory listing, etc.
- o Program invocation and control: create process, terminate process, etc.
- o Input/Output: List file etc.
- o System Operation: Starting the system, monitoring its components, etc.

Each of the following sections presents a scenario from one of these categories.

13.2 Registering a New User

New users may be added to the system only by members of the administrative group. The command to create a principal entry issues an Invoke operation specifying the logical name for the principal data base manager (CL_Principal) as the target process, and including the Create_Principal operation and its parameters in the message text. The Invoke uses the Locate(CL_Principal) operation, to find an available principal data base manager, then sends the message text to one of the sites that responds using SendToHost. The site identifier may be cached to simplify subsequent requests. The principal data base manager creates a user entry and returns the unique identifier for the new object. This UID is the Cronus internal name of the principal, and will appear in Access Group Sets and Group specifications. It may also be used to identify the user record whenever that record needs to be accessed.

When a principal is added, a number of user data base entries are initialized. One of those is the priority range authorized for the user. A private directory is created, and the principal is given all rights to it. The pathname for this directory is entered as the default home directory for the principal. The home directory serves as the repository for command interpreter profile data that specifies user-customizable system features.

13.3 Login

A user may connect to Cronus either through Telnet and a standard session agent running on a shared Cronus host, or through a Cronus Terminal Access Computer (TAC). Telnet supports access from outside the cluster through gateways, and from other devices obeying the protocol.

Access through a Cronus terminal device process is available only from a host that supports Cronus interprocess communication protocols and will probably be supported only on workstations or Cronus TACs. It is more powerful, because the access point software is fully integrated with Cronus.

To initiate a session, a user must have a terminal device process to manage his terminal communication, and a session controller process to manage interactions with the system. Telnet access requires both processes to execute on a shared host

of the system. A workstation access path can support both processes; a Cronus TAC access path places the terminal device process in the TAC and the session controller process on a shared host.

Login is handled by the Cronus session controller process. The user is prompted for a login name and password, which are used by the session controller process to build a request to the Authentication Manager by invoking the operation.

Authenticate_As(name,encrypted_password)

On receiving this message, the Authentication Manager retrieves the associated principal data base entry, verifies the password, and creates the Access Group Set for the process.

The Authentication Manager interacts with the Cronus Session Manager to record the session. The Session Manager assigns a session identifier and adds it to the table of active sessions. A session record contains the UIDs of the session principal, controller process, and terminal device process. This table is used to satisfy status requests about the cluster and active users. Some emergency procedures, (for example, destroying all processes associated with a session), may also rely upon this table.

The session identifier, the AGS, and other user data base entries are placed in the process environment through an interaction with the process manager for the authenticating process.

After modifying the process environment to indicate successful authentication, Authenticate_As returns the principal UID to the authenticated process. This identifier is used to interrogate the user data base for other information needed to complete the login sequence. One such item is the default home directory, the symbolic name of the initial Cronus directory which is used for unrooted catalog lookup operations, including the search for additional user initialization data. The directory name is converted to a catalog entry UID by an interaction with the catalog manager, and the UID is stored in the process descriptor.

A principal may have a default program registered with the Authentication Manager; if so, this program is executed at login time. If no program is specified, the standard command interpreter is assumed. The standard input and output for the executing process are directed to the principal's terminal device

process.

13.4 Accessing a File

Each process descriptor contains (among other things) an entry for the UID of the current directory. This value is initialized at login to the principal's home directory, but can be modified during the course of the session. The current directory is inherited by a new program carrier process.

Suppose a client process wants to read the first 500 bytes of data in the primal file with the symbolic name :a:b:c. To do this, it would obtain the UID for the Primal File by means of:

```
Lookup(nullDirUID, ":a:b:c", true)
-> abDirUID, abcCatEntUID, abcCatEntContents.
```

By convention, the UID for the null directory, nullDirUID, is used to specify the starting directory whenever a complete name is to be looked up. Next, it would read the file data by means of:

```
Read(abcCatEntContents.ObjectUID, 0, 500)
```

which would cause the primal file manager to send the first 500 bytes of data for the file.

These operations are made available by a single function call in the Process Support library.

```
ReadFileData(":a:b:c", 0, 500)
```

Now, assume that a process has a relative symbolic name for a file. The current directory UID is included in the request to the catalog to look up the file name. Using the general form of Invoke, the catalog manager is found based on the hint in the catalog entry UID. The catalog manager performs the lookup and returns the primal file UID associated with the symbolic name. The primal file UID is then used to find the file manager for this object, again using the hint which is part of the file UID to locate the manager.

13.5 Creating a File

A Cronus cluster may contain many hosts with file managers, each willing to store and retrieve file data at the request of other processes. The operation

Locate(CL_Primal File)

can be invoked by a process to determine the set of accessible primal file managers.

One policy for the creation of files might be to try to create the file on the same host as the creating process if a local primal file manager responded. If this is not possible, a remote manager can be selected and asked to create the file. The primal files manager include status information, information in the responses, such the amount of unused disk storage available; a measure of the current I/O and processor load; or a restriction on the principal UIDs that may to create files through this manager. This information can be used to select a storage site for the file. The selection strategies are packaged in a library routines in the Process Support Library.

The file may need a symbolic catalog entry. The catalog entry operation is carried out by the catalog manager of the directory to which the file is being added.

Suppose that the client process wants to create a file and to give it the symbolic name :a:b:c. Further suppose that a directory named :a:b already exists.

First the client would use the

Create -> FileUID

operation to create a new primal file. The file would be empty. The client could write data into the file by means of:

Write(FileUID, BytePosition, Data)

or by bracketing the write(s) by

Open(FileUID, ReadWrite, Frozen)

and

```
Close(FileUID, RetainWrites)
```

operations.

To catalog the file, the client first obtains the UID of the directory that will contain the catalog entry for the new name:

```
Lookup(nullDirUID, ":a:b", true)
  -> aDirUID, abCatEntUID, abCatEntContents
```

and then enters the new name:

```
Enter(abCatEntContents.ObjectUID, "c", FileUID)
  -> abcCatEntUID.
```

If there were no directory :a:b or :a, then the client would first have to create both :a and :a:b. This could be done as follows. First the client would obtain the UID for the root directory. By convention the name of the root directory is :Root. The fact that the root directory is cataloged in itself represents the only violation of the tree structured property of the Cronus symbolic name space.

```
Lookup(nullDirUID, ":Root", true)
  -> rootDirUID,
    rootCatEntUID,
    rootCatEntContents
```

Next, the client would create the directory :a:

```
CreateDir(rootDirUID, "a")
  -> aDirUID, aCatEntUID
```

and then, it would create the directory :a:b:

```
Create(aDirUID, "b") -> abDirUID, abCatEntUID.
```

At this point, the symbolic name :a:b:c can be established, as above, for the primal file.

The Process Support Library contains routines coupling the creation and naming of files, to avoid the situation where a failure produces a file which does not have a symbolic catalog entry and hence is not easily accessed. The operations are ordered such that the symbolic name is entered before the file is closed. If the process fails after the name is entered, the catalog entry may be deleted by explicit user commands, or by

automatic recovery mechanisms.

13.6 Deleting a File

Suppose the name of the file to be deleted is >a>b>c. Deletion is accomplished by the following operations:

```
Lookup(nullDirUID, ":a:b:c", true)
  -> abDirUID, abcCatEntUID, abcCatEntContents
```

```
Delete(abcCatEntContents.ObjectUID)
```

```
Remove(abcCatEntUID)
```

If the primal file and catalog manager are coupled, the Delete operation could have the side effect of invoking the Remove operation.

13.7 Listing a Symbolic Catalog Directory

Suppose the name of the directory is :a:b:c. A utility program executes the following sequences of operations to print the desired file names.

```
InitScan(nullDirUID, ":a:b:c:*.**")
  -> abcScanState,
      xDirUID,
      xCatEntUID,
      xCatEntContents
```

```
repeat until abcScanState indicates end of scan
[ if TypeOf(xCatEntContents.ObjectUID) = A_filetype
  then print xCatEntContents.SymbolicName;
```

```
ScanDirectory(abcScanState)
  -> abcScanState,
      xDirUID,
      xCatEntUID,
      xCatEntContents;
```

```
]
```

13.8 Running a Program

Application programs are executed within program carrier objects. The creation of an application process has three steps: a program carrier is created, the program carrier is loaded with the program image, and the program carrier is started.

The program image will generally be obtained from a Cronus file, which may be anywhere within the Cronus file system. A routine, that combines these process creation steps process creation will be available in the PSL. This routine takes as one of its arguments the symbolic name of the program image file. The symbolic name is translated to the file UID by means of a symbolic catalog lookup, and the file UID is used to load the program image into a new program carrier object.

In a heterogeneous system, a particular program image can only be executed on certain processors. A VAX program image, for example, can only be executed on a VAX host. Some mechanism must exist to match the the program image to a processor capable of executing it.

Subtypes of program carriers are defined for each processor architecture for example, CT_VAX_Program_Carrier. These subtypes contribute no new operations to objects of type CT_Program Carrier, but provide a means of locating a specific kind of processor. For example, the operation

Locate(CL_VAX_Program_Carrier)

will attempt to locate all program carrier managers on VAX hosts.

Executable files are subtypes of primal files with the type CT_Executable. The descriptor of a program image file contains the logical name of a program carrier subtype, e.g., CL_VAX_Program_Carrier. The file descriptor may also contain other information such as special host requirements. An operation on program carrier managers, Resource_Test, determines if a particular manager has the resources which are prerequisites to execution; the Create_Process routine can invoke this test whenever a process has special needs.

The actions carried out by the library routine can now be described in greater detail:

1. The symbolic program name is translated to an executable file UID, by means of a symbolic catalog lookup.

2. The routine requests the file descriptor of the program image file, by invoking the Read_Descriptor on the file object.
3. The required program carrier type and any special requirements are determined from the file descriptor.
4. A Locate operation finds the Program Carrier Managers capable of executing this process, and a Resource_Test operation narrows the candidates further.
5. A Program Carrier Manager is selected according to some policy (18) and the operation Create_Program_Carrier is invoked on it; the UID of the new Program Carrier object is returned.
6. The Load_Program operation is invoked on the program carrier object.
7. When the load operation is complete, the routine receives a reply from the Program Carrier object, and then invokes Proceed on the Program Carrier to start it.

The Create_Program_Carrier operation takes as a implicit parameter the process descriptor of the creating process, which is inherited (with certain changes) by the new process.

A process descriptor contains some information which is maintained securely by the system (e.g., the process UID, and the UID of its principal) and an open-ended set of information inserted into the descriptor by the Change_Process_Descriptor operation. All of the open-ended information is inherited directly by the descendants of the process. Some of the system information is inherited (e.g., the principal is normally inherited) and some of it is not (e.g., the process UID of a descendant is unique to it). The system information defines the authority of the new process for access to information and resources.

The creating process may invoke Change_Process_Descriptor

(18) A reasonable policy might select the Program Carrier manager on the local host, if it is a candidate, and to select the most lightly loaded host (from information in the reply to Locate) if it is not. Many other policies are possible, and exploring the possibilities is an important area of future work.

after but before starting, the program carrier to make changes in the descriptor.

13.9 Starting a Cronus Service

In this section we sketch a scenario which might be directed by a cluster control station, to startup, operate, and take down a Time Service instance on one host. It is indicative of the steps required to initiate and control an initial process load sequence. The steps required to bring up each host to the point assumed in this scenario have been discussed in Section 12.

The Cronus Time Service has two main functions:

1. To respond to direct requests for the date and time, and for format conversions among the Cronus date and time formats.
2. To periodically multicast the date and time on a well-known VLN multicast channel.

Assume that host CVAX has joined the Cronus system, and the primal process manager is the only active Cronus process. The control station performs

```
InvokeOnHost("CVAX",  
             CL_Primal_Process,  
             <(CK_Operation_Name,CO_Service_List)>  
)
```

and receives in reply a list of the services which could be created on CVAX; only the PPM is marked as active. The logical name CL_Time_Service is contained in the list. The control station then performs

```
InvokeOnHost("CVAX",  
             CL_Primal_Process,  
             <(CK_Operation_Name,CO_Create_Primal_Process)  
             (CK_UID_Service_Name,CL_Time_Service)>)
```

The Time Service process is created and started, and the control station receives a reply containing CVAX_Time_Service_UID, the specific UID of the Time Service Primal Process. The Time

Service begins its work, and if left undisturbed will periodically multicast the date and time forever. The control station (or any other Cronus process) could request the current date and time by performing

```
InvokeOnHost("CVAX",  
  CL_Time_Service,  
  <(CK_Operation_Name,CO_Date_Time)>)
```

At some later time, it becomes necessary to temporarily inhibit the periodic multicasts of the Timer Service. The control station performs

```
InvokeOnHost("CVAX",  
  CVAX_Time_Service_UID,  
  <(CK_Operation_Name,CO_Change_Process_Descriptor),  
  (CK_Modify,) (CK_IPCEnabled,"false")>)
```

After the control station receives the reply confirming this operation, it is known that all IPC to or from the Time Service has been inhibited. The Time Service process continues to exist, however, and is eventually restored to its normal function when the control station performs

```
InvokeOnHost("CVAX",  
  CVAX_Time_Service_UID,  
  <(CK_Operation_Name,CO_Change_Process_Descriptor),  
  (CK_Modify,) (CK_IPCEnabled,"true")>)
```

Finally, perhaps in preparation for replacing the Time Service code with a new version, the control station does

```
InvokeOnHost("CVAX",  
  CVAX_Time_Service_UID,  
  <(CK_Operation_Name,CO_Destroy)>)
```

and the Time Service process is known to be destroyed when the reply arrives at the control station.

14 Cronus Primal System Support

14.1 Primal System Hardware

The Advanced Development model of the Cronus distributed operating system will have three mainframe computers, four GCEs, and a gateway. The mainframe computers are two BBN C70s and a Digital Equipment Corporation VAX 11/750, the GCEs are Multibus computers with M68000 central processors, and the gateway is an DEC LSI-11 based computer.

The C70 computers are configured as general development machines. The first, C70-1, is the site of the majority of the development work since it supports both the C70 development tools and those of the GCEs. We will rent time on a second C70, C70-2, which will be used to exercise Cronus support for reliable redundant hosts, and to test scalability. Both C70s will run UNIX version 7 as released by BBN Computer Corporation and modified by the Cronus project.

The VAX 11/750 provides a VMS-based software development environment, as well as a mainframe of a distinctly different architecture. Its purpose in the ADM is to provide a limited integration host. Since it is a large well-supported mainframe, it will contain its own development environment, and we will also use it as a source of computer power for general tasks, both to off-load the C70, and to test real usage of the Cronus heterogeneous host environment. The VAX is configured to reflect its usage as a software development machine.

The Cronus system has four GCEs, configured for a variety of tasks. Since they are compatible machines, their configurations will vary over time, as we perform different experiments on the network, and as we make board substitutions to make one GCE perform functions of another which is temporarily out of service. The configuration table for the GCEs should be regarded as only a typical set of GCE configurations.

The Cronus gateway is implemented on an DEC LSI-11 computer. This would normally be a task for a GCE; however, standard internet gateways are currently implemented on LSI-11, and adoption of the LSI-11 gateway allows us to obtain an off-the-shelf implementation. The next generation of internet gateways is expected to be built on M68000 computers, and at that time we will probably move the gateway to a GCE.

C70-1	1 Mbyte main storage 2 80 Mbyte removable disk drives Magnetic Tape Drive, 800/1600 bpi, 125 ips (Cipher) Arpanet 1822 LHDH interface Ethernet interface (using Interlan protocol module)
C70-2	1/2 Mbyte main storage 2 160 Mbyte removable disk drives Arpanet 1822 LHDH interface Ethernet interface (using Interlan protocol module)
VAX 11-750	1 Mbyte main memory 1 160 Mbyte Winchester disk Magnetic tape drive, 1600 bpi, 40 ips MDI high speed synchronous serial interface 3COM Ethernet Interface VMS Operating System

Table 4. Software Development Hosts

GCE-1+2 Forward Technology M68000
processor with 256 Kbytes memory
Micro-Memory 256 Kbyte memory board
80 Mbyte Winchester Disk Drive and SMD interface
3COM Ethernet Interface
9-slot Multibus backplane

GCE-3 Forward Technology M68000 processor
with 256 Kbytes memory
Micro-Memory 256 Kbyte memory board
8-line RS-232 serial interface
3COM Ethernet Interface
9-slot Multibus backplane

GCE-4 Forward Technology M68000 processor
with 256 Kbytes memory
Micro-Memory 256 Kbyte memory board
8-line RS-232 serial interface
300 lpm line printer
3COM Ethernet Interface
9-slot Multibus backplane

Table 5. Generic Computing Elements -- Typical Configurations

Gateway LS11/03 processor card
64 Kbyte memory card
DLV11J 4 line terminal card
MRV11C ROM card (bootstrap)
ACC 1822 interface with DMA
Interlan NI2010 QBUS Ethernet controller
BBN FNV11 Fibernet interface
MDB backplane and power-supply.

Table 6. Gateway Configuration

14.2 Virtual Local Network

14.2.1 Purpose and Scope

The Cronus Virtual Local Network (VLN) provides interhost message transport in the Cronus Distributed Operating System. The VLN client interface is available on every Cronus host. Client processes can send and receive messages using specific, broadcast, or multicast addressing.

The VLN stands in place of a direct interface to the physical local network (PLN). This additional level of abstraction is defined to meet two major system objectives:

- o Compatibility. The VLN is compatible with the Internet Protocol (IP) and with higher-level protocols, such as the Transmission Control Protocol (TCP), based on IP.
- o Substitutability. Cronus software built above the VLN is dependent only upon the VLN interface and not its implementation. It is possible to substitute one physical local network for another provided that the VLN interface specification is satisfied.

This description assumes the reader is familiar with the concepts and terminology of the DARPA Internet Program; reference [NIC 1982] is a compilation of the important protocol specifications and other documents. Documents in [NIC 1982] of special significance here are [Postel 1981a] and [Postel 1981b].

The Advanced Development Model ADM will be connected to the ARPANET, and it is important that the ADM conform to the standard and conventions of the DARPA internet community. In addition, a large body of software has evolved, and continues to evolve, in the internet community. For example, protocol compatibility permits Cronus to assimilate existing software components providing electronic mail, remote terminal access, and file transfer.

The substitutability goal reflects the belief that different instances of Cronus will use different physical local networks. Substitution may be desirable for reasons of cost, performance, or other properties of the physical local network such as mechanical and electrical ruggedness.

Figure 15 shows the position of the VLN in the lowest layers of the Cronus protocol hierarchy. The VLN interface specification leaves programming details of the interface and

host-dependent issues unspecified. The precise representation of the VLN data structures and operations will vary from machine to machine, but the functional capabilities of the interface are the same regardless of the host.

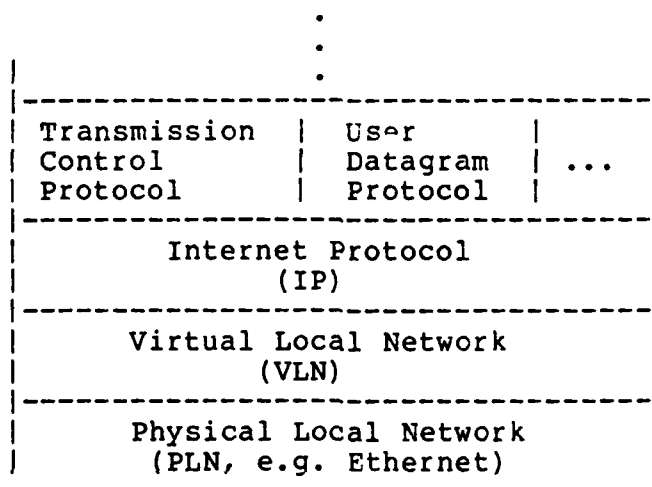


Figure 15 . Cronus Protocol Layering

The VLN is completely compatible with the Internet Protocol as defined in [Postel 1981b]. No changes or extensions to IP are required to implement IP above the VLN.

14.2.2 The VLN-to-Client Interface

The VLN layer provides a datagram transport service among hosts in a Cronus cluster, and between these hosts and other hosts in the DARPA internet. The hosts belonging to a cluster are attached to the same physical local network. Communication with hosts outside the cluster is achieved through internet gateways, shown in Figure 16, connected to the cluster. The VLN routes datagrams to a gateway if they are addressed to hosts outside the cluster, and delivers incoming datagrams to the

appropriate VLN host. A VLN is a network in the internet, and thus has an internet network number(19).

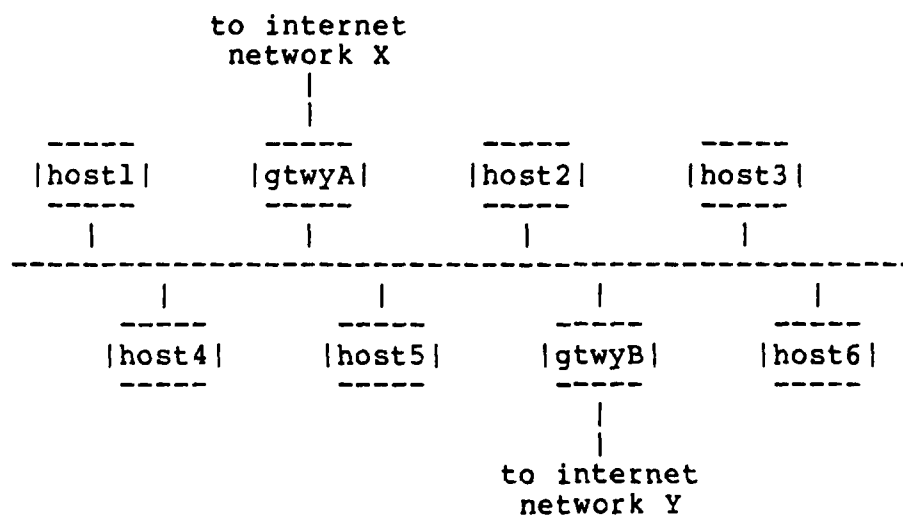


Figure 16 . A Virtual Local Network Cluster

The VLN interface will have one client process on each host, normally the host's IP implementation. The VLN performs no multiplexing/ demultiplexing function.

The structure of messages which pass through the VLN is identical to the structure of internet datagrams. The VLN definition assumes that there is a well-defined representation for internet datagrams on any host supporting the VLN interface. The argument name "Datagram" in the VLN operation definitions below refers to this well-defined but host-dependent datagram representation.

(19). The network numbers for the PLN and VLN may be the same or different. If the numbers are different, the gateways are somewhat more complex. Either approach is consistent with the internet model.

The VLN guarantees that a datagram of 576 or fewer octets can be transferred between any two VLN clients. Although larger datagrams may be transferred between some client pairs, clients should avoid sending datagrams exceeding 576 octets unless there is clear need to do so. The sender must be certain that all hosts involved can process the oversized datagrams.

The internal representation of an VLN datagram is not included in the specification, and may be chosen for implementation convenience or efficiency.

Although the structure of internet and VLN datagrams is identical, the VLN-to-client interface places its own interpretation on internet header fields, and differs from the IP-to-client interface in significant respects:

1. The VLN layer uses only the Source Address, Destination Address, Total Length, and Header Checksum fields in the internet datagram; other fields are accurately transmitted from the sending to the receiving client.
2. Internet datagram fragmentation and reassembly is not performed in the VLN layer, nor does the VLN layer implement any aspect of internet datagram option processing.
3. At the VLN interface, a special interpretation is placed upon the Destination Address in the internet header, which allows VLN broadcast and multicast addresses to be encoded in the internet address structure.
4. With high probability, duplicate delivery of datagrams sent between hosts on the same VLN does not occur.
5. Between two VLN clients S and R in the same Cronus cluster, the sequence of datagrams received by R is a subsequence of the sequence sent by S to R; a stronger sequencing property holds for broadcast and multicast addressing.

In the DARPA internet, an internet address is defined to be a 32-bit quantity that is partitioned into two fields, a network number and a local address. VLN addresses share this basic structure, but it attaches special meaning to the local address field of a VLN address.

Each network is assigned a class (A, B, or C), and a network number. The partitioning of the 32-bit internet address into

network number and local address fields as a function of the class of the network is shown in Table 7.

	Width of Network Number	Width of Local Address
Class A	7 bits	24 bits
Class B	14 bits	16 bits
Class C	21 bits	8 bits

Table 7. Internet Address Formats

The bits not included in the network number or local address fields encode the network class, e.g., a 3 bit prefix of 110 designates a class C address (see [Postel 1981a]).

The interpretation of the local address field is the responsibility of the network. For example, in the ARPANET the local address refers to a specific physical host. VLN addresses, in contrast, may refer to all hosts (broadcast) or groups of hosts (multicast) in a Cronus cluster, as well as specific hosts inside or outside of the cluster. Specific, broadcast, and multicast addresses are all encoded in the VLN local address field (20). The meaning of the local address field of a VLN address is defined in Table 8.

(20). The ability of hosts outside a Cronus cluster to transmit datagrams with VLN broadcast or multicast destination addresses into the cluster may be restricted by the cluster gateway(s), for reasons of system security.

<u>Address Modes</u>	<u>VLN Local Address Values</u>
Specific Host	0 to 1,023
Multicast	1,024 to 65,534
Broadcast	65,535

Table 8. VLN Local Address Modes

In order to represent the full range of specific, broadcast, and multicast addresses in the local address field, a VLN network should be either class A or class B.

The VLN does not attempt to guarantee reliable delivery of datagrams, nor does it provide negative acknowledgements of damaged or discarded datagrams. It does guarantee that received datagrams are accurate representations of transmitted datagrams.

The VLN guarantees that datagrams will not replicate during transmission, so each intended receiver, a given datagram given to the VLN by higher levels is received once or not at all(21).

Between two VLN clients S and R in the same cluster, the sequence of datagrams received by R is a subsequence of the sequence sent by S to R, that is datagrams are received in order, possibly with omissions. A stronger sequencing property holds for broadcast and multicast transmissions. If receivers R1 and R2 both receive broadcast or multicast datagrams D1 and D2, either they both receive D1 before D2, or they both receive D2 before D1.

While a VLN could be implemented on a long-haul or virtual-circuit-oriented PLN, these networks are generally ill-suited to the task. The ARPANET, for example, does not support broadcast or multicast addressing modes, nor does it provide the VLN sequencing guarantees. If the ARPANET were the base for a VLN

(21). A protocol operating above the VLN layer (e.g., TCP) may employ a retransmission strategy; the VLN layer does nothing to filter duplicates arising in this way.

implementation, broadcast and multicast would have to be constructed from specific addressing, and a network-wide synchronization mechanism would be required to implement the guarantees. Although the compatibility and substitutability benefits might still be achieved, the implementation would be costly, and performance poor.

A good implementation base for a Cronus VLN would be a high-bandwidth local network with all or most of these characteristics:

1. The ability to encapsulate a VLN datagram in a single PLN datagram.
2. An efficient broadcast addressing mode.
3. Natural resistance to datagram replication during transmission.
4. Sequencing guarantees like those of the VLN interface.
5. A strong error-detecting code (datagram checksum).

Good candidates include Ethernet, the Flexible Intraconnect, and Pronet, among others.

14.2.3 A VLN Implementation Based on Ethernet

The Ethernet local network specification is the result of a collaborative effort by Digital Equipment Corp., Intel Corp., and Xerox Corp. The Version 1.0 specification [DEC 1980] was released in September 1980. Useful background information on the Ethernet internet model is supplied in [Dalal 1981].

The addresses of specific Ethernet hosts are arbitrary 48-bit quantities, not under the control of the DOS. The VLN implementation must map VLN addresses to specific Ethernet addresses. The mapping can not be maintained manually in each VLN host, because manual procedures are too cumbersome and error-prone for a local network with many hosts, each of which may join and leave the network frequently. A protocol is described below which allows a host to construct the mapping dynamically, beginning only with knowledge of its own VLN and Ethernet host addresses.

An internet datagram is encapsulated in an Ethernet frame by

placing the internet datagram in the Ethernet frame data field, and setting the Ethernet type field to "DoD IP", as shown in Figure 9.

The Ethernet octet ordering is required to be consistent with the IP octet ordering. If $IP(i)$ and $IP(j)$ are internet datagram octets and $i < j$, and $EF(k)$ and $EF(l)$ are the Ethernet frame octets which represent $IP(i)$ and $IP(j)$ once encapsulated, then $k < l$. Bit orderings within octets must also be consistent.

Each VLN component maintains a virtual-to-physical address map (the VMap) which translates a 32-bit specific VLN host address to a 48-bit Ethernet address. The VMap data structure and the operations on it will be implemented using hashing techniques.

Each host controller has an Ethernet host address (EHA) to which it responds. The EHA is determined by Xerox and the controller manufacturer. In addition, the VLN assigns a multicast-host address (MHA) to each host. This multicast address is constructed from the local host portion of the internet address.

When the VLN client sends a datagram to a specific host, the local VLN component encapsulates it and transmits it without delay. The Source Address in the Ethernet frame is the EHA of the sending host. The Ethernet Destination Address is formed from the destination VLN address in the datagram, and is either:

- o the EHA of the destination host, if the sending host knows it, or
- o the MHA formed from the host number in the destination VLN address, as described above, if the sending host does not know the EHA corresponding to the host number.

When a VLN component receives an Ethernet frame with type "DoD IP", it decapsulates the internet datagram and delivers it to its client. If the frame was addressed to the EHA of the receiving host, no further action is taken. If the frame was addressed to the MHA of the receiving host, the VLN component broadcasts an update for the VMaps of the other hosts. The

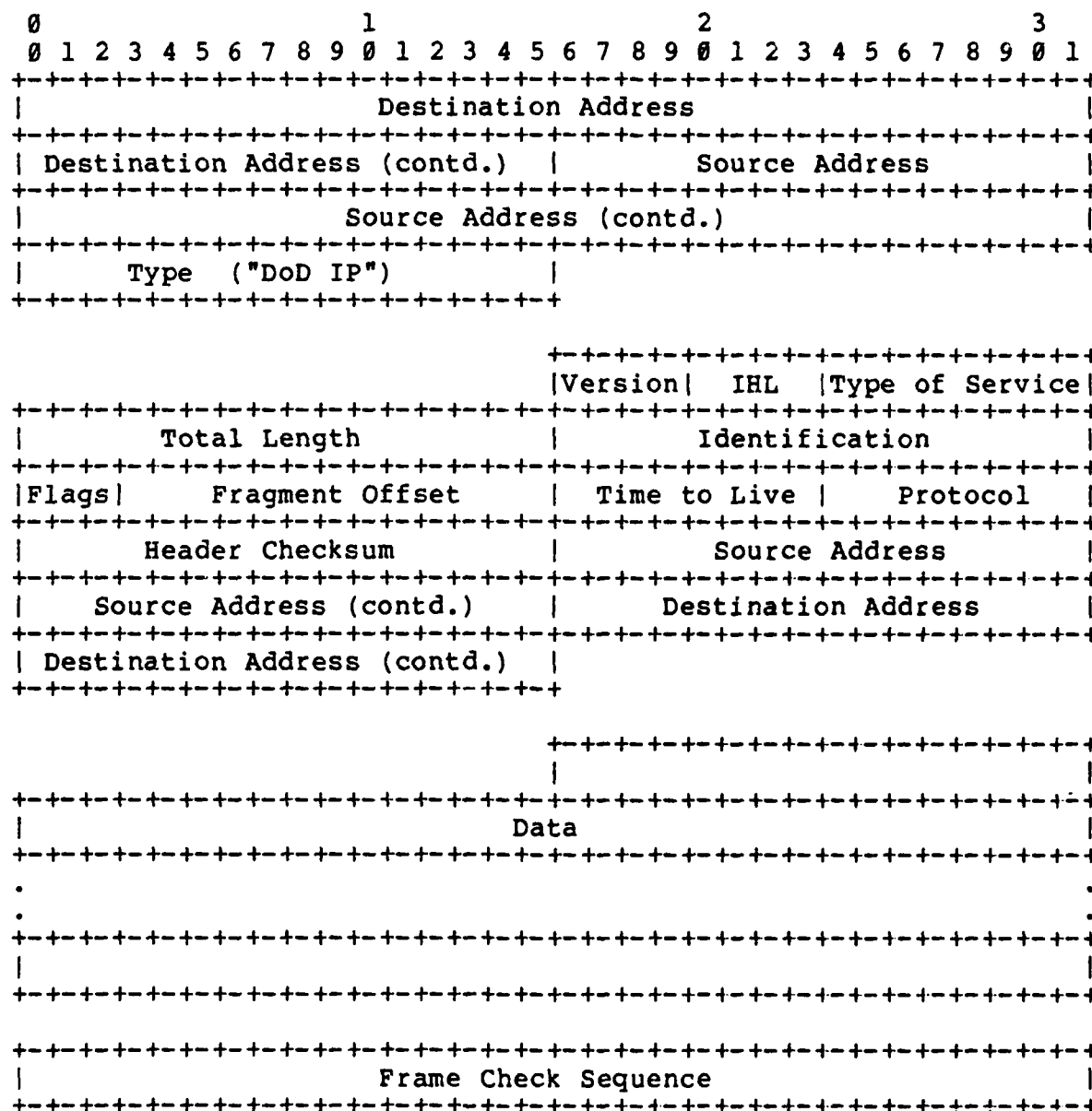


Table 9. An Encapsulated Internet Datagram

other hosts can then use the EHA of this host for future traffic. If the MHA is represented as a sequence of octets in hexadecimal, it has the form:

A B C D E F
09-00-08-00-hh-hh

A is the first octet transmitted, and F the last. The two octets E and F contain the host local address:

E F
000000hh hhhhhhhh
^ ^
MSB LSB

The type field of the Ethernet frame containing the update is "Cronus VLN", and the format of the data octets in the frame is:

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
Subtype ("Mapping Update")										Host VLN Address																													
Host VLN Address (contd.)																																							

When a local VLN component receives an Ethernet frame with type "Cronus VLN" and subtype "Mapping Update", it performs a StoreVPPair operation using the Ethernet Source Address field and the host VLN address sent as frame data.

A VLN datagram will be transmitted in broadcast mode if the specifies the VLN broadcast address (local address = 65,535, decimal) as the destination. The receiving VLN component merely decapsulates and delivers the VLN datagram.

The implementation of multicast addressing is more complex. Each host defines the number of multicast addresses which can be simultaneously "attended" (listened to). This number is a function of the particular Ethernet controller hardware and of the resources that the host dedicates to multicast processing. The VLN protocol permits a host to attend any number of multicast

addresses, from 0 to 64,511 (the entire VLN multicast address space), independent of the controller in use.

It is possible to implement the VLN multicast mode using only the Ethernet broadcast mechanism. Every VLN host would receive and process every VLN multicast, discarding uninteresting datagrams. More efficient operation is possible if some Ethernet multicast addresses are used, and if the Ethernet controller has multicast recognition which automatically discards misaddressed frames.

There is no standard for multicast recognition. The 3COM Model 3C400 controller performs no multicast address recognition. It passes all multicast frames to the host for further processing. The Intel Model iSBC 550 controller permits the host to register a maximum of 8 multicast addresses with the controller, and the Interlan Model NML0 controller permits a maximum of 63 registered addresses.

A VLN-wide constant, Multicast_Registered, is equal to the smallest number of Ethernet multicast addresses that can be simultaneously attended by all hosts in the VLN. A network composed of hosts with the Intel and Interlan controllers mentioned above, for example, would have Multicast_Registered equal to 7 (22); a network composed only of hosts with 3COM Model 3C400 controllers would have Multicast_Registered equal to 64,511, since the controller itself does not restrict the number of Ethernet multicast addresses to which a host may attend (23).

A mapping is defined which translates the VLN multicast address to an Ethernet multicast address. The first Multicast Registered VLN multicast addresses are assumed to be attended by each host. The local address portion of the internet address of a VLN multicast channel is a decimal integer M in the range 1,024 to 65,534.

1. $(M - 1,023) \leq \text{Multicast_Registered}$. In this case, the Ethernet multicast address is

09-00-08-00-mm-mm

2. $(M - 1,023) > \text{Multicast_Registered}$. The Ethernet broadcast (22); Multi_Registered is 7, rather than 8, because one multicast slot in the controller is reserved for the host's MHA. (23). For the Cronus Advanced Development Model, Multicast Registered is currently defined to be 60.

address is used. A VLN component which attends VLN multicast addresses in this range must receive all broadcast frames, and select those with VLN destination address corresponding to the attended multicast address.

Delivered datagrams are accurate copies of transmitted datagrams because VLN components do not deliver datagrams with invalid Frame Check Sequences. A 32-bit CRC error-detecting code is applied to Ethernet frames.

Datagram duplication does not occur because the VLN layer does not perform retransmissions, the primary source of duplicates in other networks. Ethernet controllers do perform retransmission as a result of collisions on the channel, but the collision enforcement mechanism or "jam" assures that no controller receives a valid frame if a collision occurs.

The sequencing guarantees hold because mutually exclusive access to the transmission medium defines a total ordering on Ethernet transmissions, and because a VLN component buffers all datagrams in FIFO order.

14.2.4 VLN Operations

There are seven functions defined at the VLN interface. An implementation of the VLN interface has wide latitude in the presentation of these operations to the client; for example, the functions may or may not return error codes.

The functions are to occur synchronously or asynchronously with respect to the client's computation. We expect that the `ResetVLNInterface`, `MyVLNAddress`, `SendVLNDatagram`, `PurgeMAddresses`, `AttendMAddress`, and `IgnoreMAddress` operations will be synchronous with respect to the client. `ReceiveVLNDatagram` will usually be asynchronous; that is, the client initiates the operation, continues to compute, and at some later time is notified that a datagram is available.

`ResetVLNInterface()`

The VLN for this host is reset. For the Ethernet implementation, the operation `ClearVPMAP` is performed, and a frame of type "Cronus VLN" and subtype "Mapping Update" is broadcast. This operation does not affect the set of attended VLN multicast addresses.

MyVLNAddress()

Returns the VLN address of this host.

SendVLNDatagram(Datagram)

When this operation completes, the VLN layer has copied the Datagram. The transmitting process cannot assume that the message has been delivered when SendVLNDatagram completes.

ReceiveVLNDatagram(Datagram)

When this operation completes, Datagram is a representation of a VLN datagram which has not previously received.

PurgeMAddresses()

When this operation completes, no VLN multicast addresses are registered with the local VLN component.

AttendMAddress(MAddress)

If this operation returns True then MAddress, which must be a VLN multicast address, is registered as an alias for this host, and messages addressed to MAddress by VLN clients will be delivered to the client on this host.

IgnoreMAddress(MAddress)

When this operation completes, MAddress is not registered as a multicast address for the client on this host.

Whenever a Cronus host comes up, ResetVLNInterface and PurgeMAddresses are performed on the VLN. A VLN component may depend upon state information obtained dynamically from other hosts, and there is a possibility that incorrect information might enter a component's state tables. A cautious VLN client could call ResetVLNInterface periodically to force the VLN component to reconstruct the tables.

A VLN component will limit the number of multicast addresses to which it will simultaneously attend; if the client attempts to register more addresses than this, AttendMAddress will return False with no other effect.

The VLN layer does not guarantee buffering for datagrams at either the sending or receiving host(s). It does guarantee that a SendVLNDatagram function performed by a VLN client will eventually complete; this implies that datagrams may be lost if buffering is insufficient and receiving clients are too slow.

14.3 Generic Computing Element Operating System

One of the more important Cronus hardware components is the Generic Computing Element (GCE). Prior to its introduction to the Cronus DOS project, CMOS was under development at BBN as a real-time operating system for several types of communication processors, such as gateways and network terminal concentrators. In addition, a support environment for building and debugging CMOS applications is available under UNIX. CMOS provides the following basic operating system features:

- o multiple processes
- o interprocess communication/coordination
- o asynchronous I/O
- o memory allocation
- o system clock management

CMOS is an open operating system; that is, no distinct division exists between the operating system and the application program. The operating system is a collection of library routines that can be easily extended by adding new routines and can be reduced by excluding unneeded routines. The programmer can directly access lower-level interfaces.

CMOS is a portable operating system. The use of the high-level language C is the principal factor in CMOS portability. Small size and simplicity are other important factors. The design minimizes the amount of machine-dependent code and segregates it. The I/O system design allows for easy replacement of device-dependent modules.

The debugging environment is provided by XMD, a display oriented debugger based on the PEN editor. All of the features of the editor are available to the user in addition to the debugger specific commands. PEN is a multi-window editor with capabilities for manipulating multiple files and edit buffers. XMD displays a special configuration of windows that are appropriate to debugging. This configuration consists of a source

window, a register display window, a breakpoint window, and a window for displaying variables.

A low-level debugger is resident in the target processor to interpret and execute commands sent to it over the communication path, currently a terminal line to the C70 UNIX host processor where XMD is running.

Access to networks will be provided to CMOS applications from three levels. At the highest level, the user can open a TCP stream. The first application at this level will be Telnet and terminal concentration software. At the next level, there is an internet datagram service. This will be used to implement inter-process communication between hosts, as well as other standard internet protocols. The lowest level is the Ethernet local network interface.

The communication module in XMD will be changed to use the Ethernet instead of a terminal line, increasing its flexibility and usefulness. Downloading will be possible over the network, plus it will be easier to debug multiple GCEs from one site.

The internal device structure was changed to give the I/O system more flexibility in dealing with the number of possible relationships between hardware devices and the interrupts generated by those devices. Without this change, the capability of writing simple device drivers for CMOS is compromised.

A name service capability was added for the run-time binding of string names to processes and devices. The name space is hierarchical and there is a notion of absolute and relative pathnames. In the presence of some form of mass storage, the names can be made non-volatile.

14.4 Cronus Utilities

14.4.1 General

A number of Cronus processes or services are so widely used or needed, that they warrant description as utilities for the system.

14.4.2 Elementary File System

14.4.2.1 Introduction

The Elementary File System (EFS) is an easily ported single host file system that serves as a common base of implementation support for Cronus file managers Cronus Generic Computing Elements (GCEs) configured with disks, on the UNIX system, and on the VAX. The underlying implementation of the EFS is constituent host dependent, but the interface presented to the Cronus File Manager is uniform. As a result, portability of the File Manager is enhanced, and the cost of integration of new hosts is reduced. The EFS was originally developed as a primitive file storage capability for the GCE mass storage devices.

The two principal design objectives of the EFS are:

1. Sufficient functional capability to support the Cronus distributed file system.
2. Simplicity and efficiency.

The principal users of the EFS will be object managers. Client processes will seldom, if ever, directly access files through the EFS. Therefore, only the most basic file operations need be supported. More complex file functions can be supported by the object managers themselves. Simple steps have been taken in the internal organization of the EFS to support effective crash recovery and system restart procedures.

The Elementary File System will have the following characteristics:

1. The name space for EFS files is flat. Names for EFS files are called FileIDs, and they are fixed length bit strings. FileIDs are not Cronus UUIDs. A FileID is unique on the EFS that generated it, but it is not unique across all Cronus hosts. The EFS is a Cronus object in much the same way that the existing UNIX or VMS file systems are Cronus objects, but
2. A EFS file is not a Cronus object.
3. File data is organized as a sequence of fixed length blocks. File i/o is sequential, and is block oriented. The basic file i/o operations are:

ReadEFSFileBlock(FileID, BlockNumber, Buffer), and
WriteEFSFileBlock(FileID, BlockNumber, Buffer).

4. There are no open or close operations. No setup is necessary to read data from or write data to an existing EFS file.
5. It is necessary to create a EFS file before writing data to it. This is accomplished by the

CreateEFSFile()

operation, which creates an empty EFS file and returns its FileID.

6. EFS files are deleted by the

DeleteEFSFile(FileID)

operation.

7. There is no access control for EFS files. Possession of the FileID for a EFS file is sufficient to access the file.

The EFS will normally be accessible only to Cronus Services. The primal file manager is an example of such a service. These services provide controlled access to the objects and operations that they implement, as described in Section 8.

In addition to supporting the local primal file manager, the EFS may be operated on as an object to permit remote access for maintenance and debugging purposes. There is a single access control list (ACL) associated with access to the entire EFS through the EFS_File Manager. Only a very few principals will be on the ACL for a EFS. An example of a principal which might be granted access to the EFS is, the "System Maintenance" principal.

14.4.2.2 File Formats

The following description of the Elementary File System structure assumes that a disk can be represented by a series of fixed length blocks. In the Cronus ADM, the storage may be:

a disk drive on a GCE;

a disk device in a UNIX system; or

a contiguous file on the VAX/VMS.

The EFS makes few demands on the underlying recording medium, and it is relatively easy to see that most potential Constituent Operating Systems will provide a construct upon which the EFS can be built.

File disk blocks are self-identifying for reliability purposes. Each block includes a header that contains the FileID and the block number. The file header in each block contains a NextBlock pointer which is the disk address of the next block, if any, in the file. The NextBlock pointer in the last block contains a special value marking the end of file.

There is a FileID Table which provides a mapping between FileIDs and the disk address of block 0 of the file (see Figure 17). The FileID Table is as a file with a well-known FileID (FileID = 1). Its block 0 will be stored at a known disk address (with an alternate copy stored at another location to prevent loss of data in case the primary block is bad). The FileID Table is a hash table.

There is a FreeDiskBlock table which records the disk blocks that are available. The FreeDiskBlock table is a bit table stored in a file with a well-known FileID (FileID = 2). Its block 0 is stored at a known disk address. When a file is deleted, its blocks are recorded in the FreeDiskBlock table, and the FileID field in the headers of each of the blocks is cleared. As disk blocks are needed they are allocated using the FreeDiskBlock table.

There are two types of EFS files. The type of the file is contained in the header of block 0. The types of EFS files are (see Figure 18):

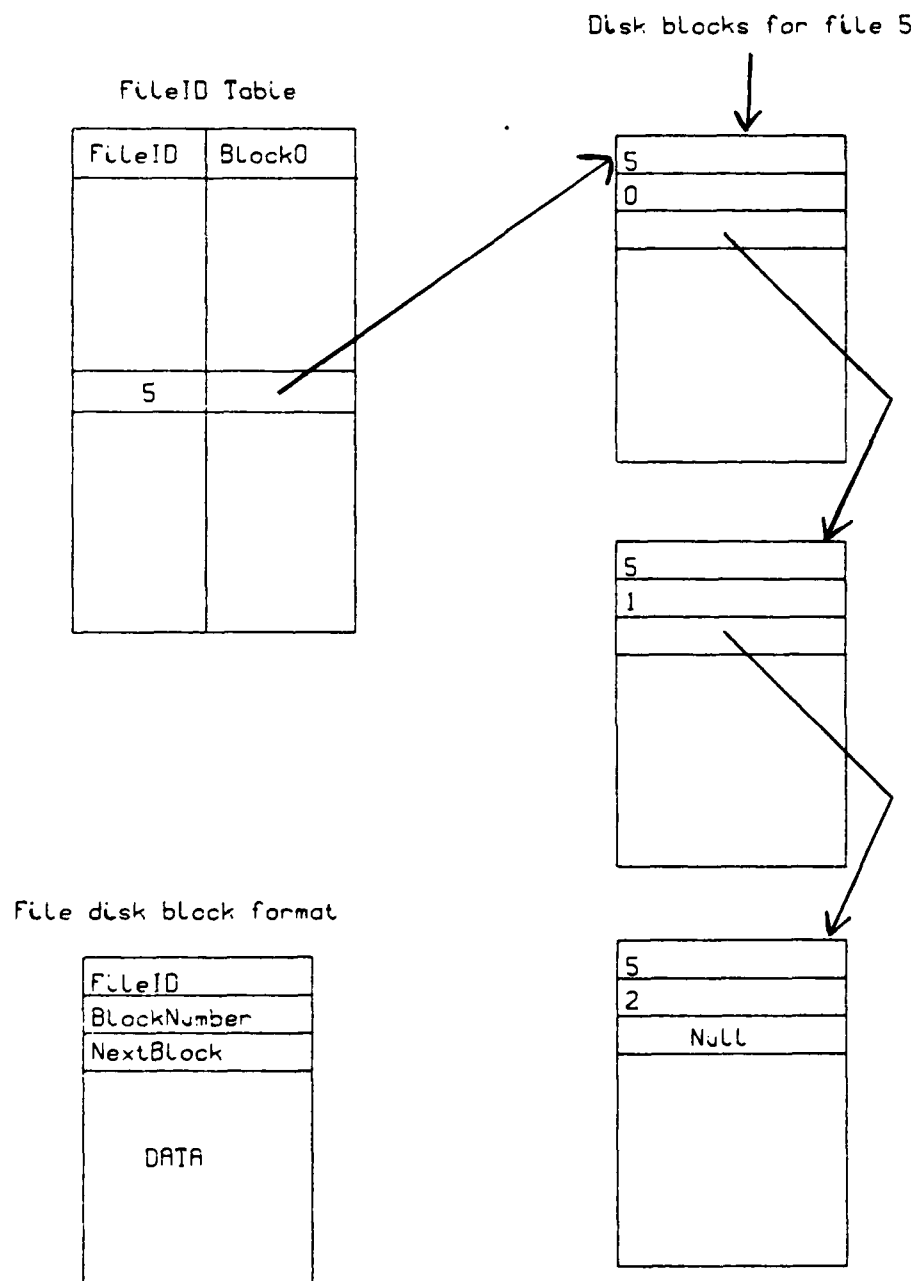
a. Short file.

This is a file, all of whose data will fit within block 0.

b. Normal file.

This is a file whose data will not fit within a single block.

A Normal file may contain index blocks which allow random access



EFS File Table
Figure 17

Random Access GCE Files

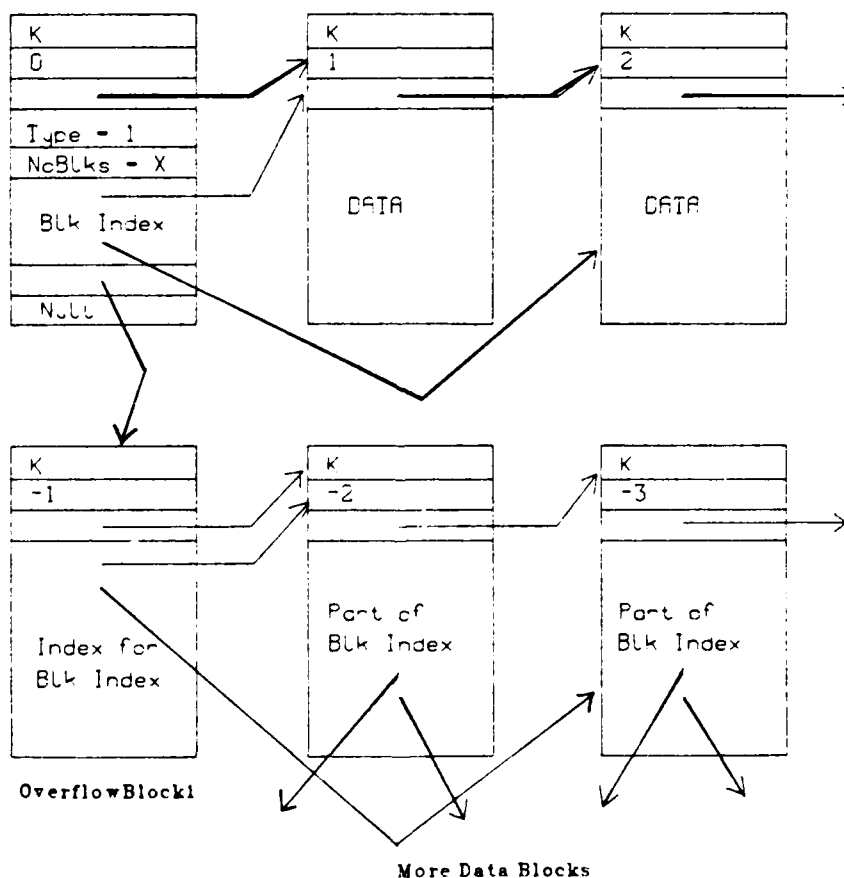
File Disk Block Format

FileID
BlockNumber
NextBlock
DATA

Small File

J
0
Null
Type - 0
DATA

Normal File



EFS File Types
Figure 18

to the file. By convention, the first of these blocks is given block number -1, and contains:

- i. A block index which holds the disk address of blocks 1 through N of the file; and
- ii. The disk addresses for two overflow blocks, named OverflowBlock1 and OverflowBlock2, which can be used to find the block index entries for blocks numbered greater than N.

If the file is very large, not all of its index will fit into block -1.

OverflowBlock1 is used as an index for blocks which store part of the block index which will not fit in block -1. Specifically, if block -1 can store indices for blocks 1 through N, if OverflowBlock1 can store M disk addresses as indices, and if each block it indexes can store P disk addresses, OverflowBlock1 can provide access to indices for $M \cdot P$ additional blocks, numbered $(N+1)$ through $(N+M \cdot P)$. The block index for the Normal file shown in Figure 18 overflows block -1 into OverflowBlock1, and is small enough that it doesn't require OverflowBlock2.

OverflowBlock2 provides an additional level of indirection for very large files. It contains an index for blocks which are used in the same manner OverflowBlock1 is. If OverflowBlock2 can hold Q disk addresses as indices, then it can provide access to indices for $M \cdot P \cdot Q$ blocks, numbered $(N+M \cdot P+1)$ through $(N+M \cdot P+1+M \cdot P \cdot Q)$.

By convention the BlockNumber for OverflowBlock1 is -2. Any index blocks referenced by OverflowBlock1, as well as OverflowBlock2 (if present), and any index blocks it references directly or indirectly are assigned BlockNumbers in a negative sequential fashion starting at -3 in the obvious manner.

Some constituent hosts will have multiple disks (in the case of UNIX, these may actually be disjoint regions on a single physical disk, and in the case of VMS, they would be multiple contiguous files). Part of the FileID specifies the disk on which the file resides. The CreateEFSFile operation takes an optional parameter which specifies a disk. If the parameter is supplied, block 0 and all subsequently created blocks of the file are allocated on the specified disk. If the parameter is not supplied, block 0 and subsequent blocks are allocated on the disk the EFS sees fit.

14.4.2.3 Disk Salvaging

There is a BadDiskBlock table which holds the disk addresses of bad disk blocks. The BadDiskBlock table is stored in a file with a well-known FileID (FileID = 3).

There is a EFS disk salvage operation which can reconstruct the FileID table, the FreeDiskBlock file, and the BadDiskBlock file, and reset the NextBlock pointers in files.

The salvager may encounter files with missing blocks. When it does, it will fill in any hole it encounters with a newly allocated filler block, linking the filler block into the file where the hole was. The FileID of the filler block will be set to the ID of the file, and its BlockNumber will be set to a special BlockNumber which identifies it as a filler block. The only data in a filler block will be the BlockNumbers of the previous and next file blocks which contain data. Higher level software can be used to recover the data in a file which contains filler blocks.

As the salvage procedure encounters bad disk blocks, it records them in the BadDiskBlock file. If it encounters a bad block which is part of a file, the salvager will remove the block from the file and substitute a newly allocated replacement block by linking it with the other blocks of the file in place of the bad block. The FileID of the replacement block will be set to the ID of the file, and its BlockNumber will be set to a special BlockNumber which identifies it is a replacement block. The only data in the replacement block will be the BlockNumber of the block it replaces. This will make it possible for higher level software to recover the data in other blocks of the file.

14.4.2.4 EFS File System Operations

The following functions will be supported by the EFS:

- o CreateEFSFile ([Disk], [NewFileID])
 -> FileID, Block0DiskAddress

Create a file, by allocating a FileID and a disk block for block 0 of the file. Make entry in the FileID Table.

For EFSs with more than one disk, the optional Disk parameter, if present, specifies the disk on which the new

file is to be stored.

If the optional NewFileID parameter is specified a check will be made to see if a file with that FileID exists. If not, NewFileID will be used as the FileID of the new file. If so, the operation will fail.

o DeleteEFSFile (FileID)

Deletes a file by: Deletion involves

1. Clearing the FileID field in each block.
2. Updating the FreeDiskBlock table; and
3. Removing the entry for the file from the FileID Table.

o ReadEFSFileBlock(FileID, BlockNumber, Buffer, [DiskAddress])

Read BlockNumber of file FileID into Buffer. Find the block by following NextBlock pointers and counting. If the FileID and BlockNumberID stored in the disk block are not the same as those specified in the call parameters, the operation fails.

The optional DiskAddress parameter is a hint. If present, the block at DiskAddress is read, and if it is block BlockNumber of file FileID, it is returned as the result of the read.

If the operation succeeds, the disk address of the block is returned.

o WriteEFSFileBlock(FileID, BlockNumber, Buffer, [DiskAddress])

Writes the data in Buffer into the specified block (BlockNumber) of the specified file (FileID). If

BlockNumber > CurrentNumberOfFileBlocks

the operation will fail. If

BlockNumber < CurrentNumberOfFileBlocks,

the disk block for BlockNumber is overwritten with the data in Buffer. If

BlockNumber = CurrentNumberOfFileBlocks 1,

a disk block is allocated and the data in Buffer is written.

The optional DiskAddress parameter is interpreted as a hint. If present and

BlockNumber < CurrentNumberOfFileBlocks,

the block at DiskAddress is read, and if it is block BlockNumber of file FileID, the data in Buffer is written into the block at DiskAddress. If

BlockNumber = CurrentNumberOfFileBlocks,

DiskAddress is ignored.

WriteEFSFileBlock is responsible for the adjustments needed for the case that the block being written converts the file into a normal file.

- o ReadRandomEFSFileDataBlock(FileID, DataBlockNumber, Buffer)

This is the random read operation. It is used to read file data blocks; index blocks are not accessible via this operation. Block -1 is used to obtain the block index (if any) for the file, and the block index is used to find the disk address of the specified file data block. The data at that block is read into Buffer.

- o WriteRandomEFSFileBlock(FileID, DataBlockNumber, Buffer)

This is the random write operation. If

DataBlockNumber > CurrentNumberOfFileDataBlocks,

then the write will fail. Block -1 is used to obtain the block index (if any) for the file, the index is used to find the disk address of the specified file data block. If

DataBlockNumber < CurrentNumberOfFileDataBlocks

the data in Buffer is written into the block specified by

the disk address. If

DataBlockNumber = (CurrentNumberOfFileDataFlocks + 1),

then a disk block is allocated for the file and the data in Buffer is written. If no block index exists when this call is made, it will create one.

o SalvageElementaryFileSystem

This initiates the salvage procedure for the EFS.

14.4.3 UNO Generation

Unique numbers are used to name Cronus objects. They may also be used for a variety of other purposes such as transaction identifiers, or cluster-wide names for objects in the application domain.

Cronus supports a service which generates unique numbers (UNOs) and is accessible to system and application processes. Processes may request a UNO at any time, from any of the hosts in a Cronus cluster. The UNO service guarantees that any requesting process is promptly supplied with a UNO. No two requests by client processes ever obtain the same UNO, over the entire lifetime of a Cronus cluster.

The UNO service is composed of two types of software components, the SmallStepper, on every Cronus host, and the LargeStepper, residing on any subset of Cronus hosts with non-volatile storage. The production of a lengthy sequence of UNOs is the result of cooperation between the SmallStepper component on a particular host and at least one LargeStepper component, sometimes remote.

Because all Cronus hosts use the UNO service, the implementation of the SmallStepper component is part of the integration cost of every host. This cost is small because the SmallStepper component is simple; the most difficult aspects of the reliability problem are treated in the design of the LargeStepper components. Delay in satisfying UNO requests is minimized because SmallStepper and LargeStepper need synchronization only infrequently; most requests can be satisfied locally and quickly.

High reliability is an important goal of the UNO generation scheme, both in the sense of continuous availability and of consistent restarts should all of the LargeStepper hosts fail or be shut down at the same time. We assume only that at least one LargeStepper host retains its non-volatile storage across an outage of all LargeStepper hosts, in order to automatically resume the production of UNOs when that host is restarted. A manual procedure exists which allows a restart of the UNO facility if all hosts lose non-volatile storage; initial UNO facility startup is a special case of this situation.

A Unique Number is a 64-bit quantity whose representation is dependent upon the host programming language and machine architecture.

The central property of the UNO is that two distinct invocations of the GenerateUNO function will never yield the same UNO. Calls to GenerateUNO by processes in different DOS clusters may yield the same bitstring; UNOs are universal only over the domain of a single cluster.

The local host number of the machine is a field of the UNO bitstring, and can be extracted with the OriginOfUNO operator. All UNOs generated by a host are strictly ordered by time of creation, and can be compared using the OrderOfUNOs operator. UNOs generated by different hosts are not comparable; OrderOfUNOs will detect and indicate this situation to its caller.

The UNO size, 64 bits, was derived from assumptions about the maximum number of UNOs needed during the lifetime of a Cronus cluster. We assume that the maximum number of hosts in a cluster is 1024, and the maximum lifetime of a DOS cluster is 100 years. The implementation strategy imposes constraints upon the rate at which UNOs can be generated (fewer than 1000 per second per host) and on the rate at which a host can leave and re-enter the cluster-wide UNO generation mechanism (about once every 10 seconds). The latter constraint increases the boot-up delay of a Cronus host by a few seconds while it initializes its SmallStepper component.

There are three primitive operations on UNOs in addition to assignment. The interface operations defined in this section are available as procedure or system calls to a client process. In the C language, assuming a typedef UNO, they might appear as follows:

```
BOOL GenerateUNO(unoptr) UNO *unoptr;
```

Generate a new UNO in the structure pointed to by unoptr and return TRUE, otherwise return FALSE.

HOSTNUM OriginOfUNO(unoptr) UNO *unoptr;

Return the internet address of the host which generated the UNO *unoptr, unless the UNO is well-known, in which case return UNDEFINED.

UNOORD OrderOfUNOs(unoptr1,unoptr2)
UNO *unoptr1, *unoptr2;

Compare the UNOs *unoptr1 and *unoptr2, and return a result indicating equality, or the ordering between the UNOS, or declare them incomparable.

These operations are continuously available, and will complete successfully unless the invoker's host fails during the call. GenerateUNO may fail (return FALSE) if all LargeStepper hosts are down or inaccessible for a long period of time. The implementation of the SmallStepper component will guarantee, that a GenerateUNO request completes in a small, bounded amount of time, unless the client's host fails during the request.

A portion of the UNO space is reserved for well-known UNOs. These will never be returned by the GenerateUNO operation; some of them are statically associated with primitive objects in the Cronus system. For these UNOs, OriginOfUNO returns the value UNDEFINED, a 32-bit quantity which is not a valid internet address. When one or more of the arguments of OrderOfUNOs is a well-known UNO, the result is UNOINCOMP.

The structure of a UNO as visible to the implementation has three fields: HostAddress, HostIncarnation, and SequenceNumber. In C, the structure might be declared:

```
typedef struct
{
    unsigned HostAddress:      10; /* bits */
    unsigned HostIncarnation:  32; /* bits */
    unsigned SequenceNumber:   22; /* bits */
}
```

A UNO with a HostIncarnation field equal to zero is a well known UNO. The HostAddress and SequenceNumber fields of a well known UNO are manually selected, arbitrary constants.

The SmallStepper component enforces mutual exclusion while responding to client requests, and while performing incarnation number updates based on transmissions from LargeStepper components.

Most invocations of GenerateUNO will cause the SmallStepper to increment a 22-bit sequence number, and combine this with the host address and current incarnation number, to form the UNO. The UNO generator obtains the host address of its host from the VLN interface.

Normally the SmallStepper maintains at least two incarnation numbers, the current incarnation number and the next incarnation number. If a GenerateUNO request causes the sequence number to overflow, the next incarnation number replaces the current incarnation number, and the sequence number is reset to zero. The next incarnation number will be refilled as soon as the SmallStepper receives a broadcast from a LargeStepper component or by incrementing a locally maintained non-volatile incarnation number. If the sequence number overflows and no next incarnation number is available, the current incarnation number becomes unavailable, and GenerateUNO will fail.

The SmallStepper component can obtain a new incarnation number passively, by listening for the next message transmitted on a well-known multicast channel if it does not maintain its own non-volatile version of it. This incarnation number becomes the current incarnation number if it was previously unavailable, or else it becomes the next incarnation number if that was unavailable, or else it is discarded. The LargeStepper components periodically transmit a new incarnation number on the channel; each number is guaranteed to be strictly greater than all previous incarnation numbers transmitted.

The separation of the SmallStepper and LargeStepper components removes the requirement for reliable, non-volatile storage at each host; the problem is now reduced to the generation of a monotone incarnation number stream by the LargeStepper components.

14.5 Process Support Library

The Process Support Library (PSL) is a collection of functions, that may be bound into the load image of a Cronus process. Only those routines actually needed by a process will be included in the load image. The data structures implemented by the PSL are within the protection domain of the process.

PSL routines are considered part of the Cronus system and will generally be maintained by system programmers. The PSL fulfills five major roles:

1. It provides a convenient interface to Cronus operations.
2. It provides access to special Cronus features such as the GenerateUNO facility and the GCE file system; these features are not normally accessed through the Operation Switch.
3. It provides the Message Structure Facility a collection of routines to build and parse messages.
4. It provides an IPC facility at a higher level than the primitive InvokeOnHost level.
5. It provides COS interface and utility routines necessary to support the production of portable programs. This includes format conversion routines and machine-dependent constants, for example.

AD-A139 983

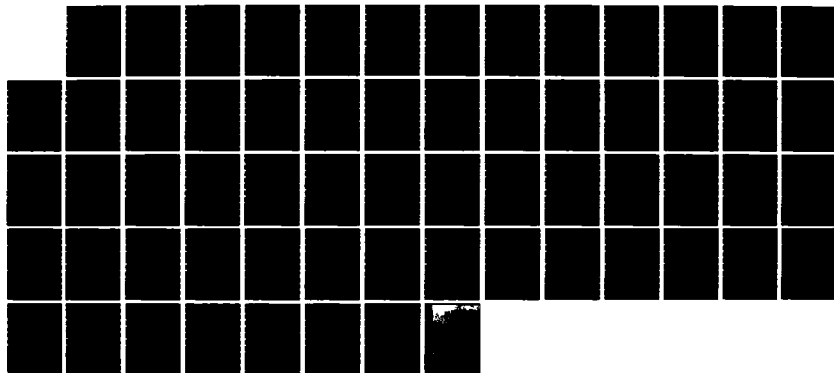
CRONUS A DISTRIBUTED OPERATING SYSTEM(U) BOLT BERANEK
AND NEWMAN INC CAMBRIDGE MA R SCHANTZ ET AL DEC 83
BBN-5261 RADC-TR-83-255 F30602-81-C-0132

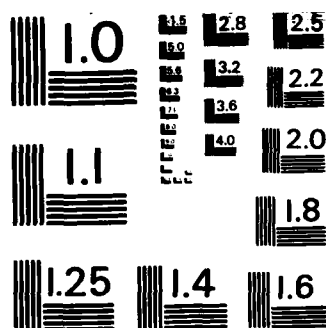
3/3

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

REFERENCES

- [BBN 5041]
"Cronus, a distributed operating system: functional definition and system concept," M. D. Hoffman, W. I. MacGregor, R. E. Schantz, & R. H. Thomas, Technical Report #5041, Bolt Beranek and Newman Inc., June 1982.
- [BBN 5086]
"Cronus, A Distributed Operating System: Interim Technical Report No. 1," R. Schantz, E. Burke, S. Geyer, M. Hoffman, A. Lake, K. Pogran, D. Tappan, R. Thomas, S. Toner, and W. MacGregor, Technical Report #5086, Bolt Beranek and Newman Inc., July 1982.
- [Dalal 1981]
"48-bit absolute internet and Ethernet host numbers," Yogen K. Dalal and Robert S. Printis, Proc. of the 7th Data Communications Symposium, October 1981.
- [DEC 1980]
"The Ethernet: a local area network, data link layer and physical layer specifications," Digital Equipment Corp., Intel Corp., and Xerox Corp., Version 1.0, September 1980.
- [NIC 1982]
"Internet protocol transition workbook," Network Information Center, SRI International, Menlo Park, California, March 1982.
- [Postel 1981a]
"Assigned numbers," Jon Postel, RFC 790, USC/Information Sciences Institute, September 1981.
- [Postel 1981b]
"Internet Protocol - DARPA internet program protocol specification," Jon Postel, ed., RFC 791, USC/Information Sciences Institute, September 1981.
- [Rentsch 1982]
"Object oriented programming," T. Rentsch, SIGPLAN Notices, September 1982, pp. 51-57.
- [Xerox 1981]
"The Smalltalk-80 system," Xerox Learning Research Group, BYTE, August 1981, pp. 36-47.

INDEX

AbortWrites.....	96
access.....	126, 162
access control.....	14, 110, 123
access control list.....	19, 94, 167
access machine.....	5
access point agent.....	126
acknowledge.....	95
acknowledgements.....	156
additive.....	17
address recognition.....	161
AddToACL.....	97
Advanced Development Model ADM.....	3, 9
application process.....	144
arc.....	99
AREYOUTHERE.....	135
arrays.....	61
ASC.....	67
asynchronous I/O.....	164
Asynchrony.....	68
atomic.....	49, 95
Attendant.....	54
AttendMAddress.....	163
authentication manager.....	19
authority.....	145
availability.....	176
available.....	177
BadDiskBlock table.....	172
basic data type.....	65
BITS.....	67
block.....	166
block index.....	171
BOOL.....	67
BOOTLOAD.....	135
BOOTYOUSELF.....	135
bound.....	13
broadcast.....	32, 154, 156
broadcast addressing mode.....	157
Buffering.....	69
buffering.....	164
byte position.....	93
C70s.....	148
cable.....	134
canonical type.....	65

Controller.....	54
controlling process.....	54
convenient.....	127
conventional features.....	22
cooperating processes.....	45
copy.....	90
COS interface.....	179
crash.....	95, 131
CRC.....	162
Create.....	45, 96
create.....	167
create a file.....	141
CreateDir.....	109
CreateEFSFile.....	167, 172
Create_Primal Process.....	50
Create_Program_Carrier.....	55
Creating a File.....	140
cronus catalog.....	25
Cronus cluster.....	3, 5
Cronus logical name.....	51
Cronus process.....	45
Cronus service.....	46, 129
Cronus sybolic service name.....	51
Cronus VLN.....	160
Cronus_Restart.....	50
ct.....	26
CT_Catalog_Entry.....	104, 107
CT_Directory.....	100, 106, 109
CT_External_Linkage.....	100, 106, 110
CT_Primal Process.....	46
CT_Program_Carrier.....	46, 52
CT_Symbolic_Link.....	100, 106
CT_Type_Name.....	27
current directory.....	100
cursor positioning.....	127
data reduction.....	132
datagram.....	9, 20, 152, 157
datagram option processing.....	154
datagram replication.....	157
debugger.....	53, 164
debugging.....	134
DebugWait.....	54
DEC LSI-11.....	148
Delete.....	94, 96
DeleteDir.....	109
DeleteEFSFile.....	173
Deleting a File.....	143
demultiplexing.....	68, 153

catalog.....	18
catalog data base.....	112
catalog manager.....	101, 112, 120
catalog the file.....	142
Cataloged Object Table.....	112, 115
cat_Lookup.....	107
cat_LookupWild.....	108
Change Process Descriptor.....	50
ChangeEntry.....	108
ChangeObjectEntries.....	109
Change_Process_Descriptor.....	48
Change_State.....	53
character strings.....	61
child.....	55
CHP.....	45
chps.....	33
class.....	154
class A.....	156
class B.....	9, 156
cleanup.....	94
Clear.....	54
Clear_Program.....	52
ClearVMap.....	162
close.....	91
Close.....	96
close.....	167
CloseAllProcessOpenFiles.....	98
CloseProcessOpenFile.....	98
CMOS.....	164
coherence.....	3, 11
coherent.....	22
collision enforcement.....	162
command interface.....	126
communication.....	33
communications.....	5
compatibility.....	20, 157
Compatibility.....	151
complex objects.....	14
connected directory.....	100
constituent host process.....	33, 45
Constituent Operation System COS.....	129
continuous.....	176
continuously.....	177
control.....	19, 129
control information.....	57
control message.....	58
control station.....	146
control traffic.....	58

Destroy.....	47
destroy.....	48
:dev.....	124
development machine.....	148
device.....	100
device ojects.....	16
devices.....	18
device-specific operations.....	124
:dev:lpt.....	124
Digital Equipment Corp.....	157
directory.....	99, 100, 112
directory objects.....	16
dispersal cut.....	114
dispersal subtree.....	114
dispersed file.....	90
distributed operating system.....	11
distribution.....	112, 114
DoD IP.....	158
download.....	134
elective keys.....	48
encapsulated.....	157
end of the file.....	97
Enter.....	107
EnterExternalLinkage.....	110
EnterLink.....	110
EntriesOf.....	109
entry name.....	99
error recovery.....	56
error-detecting code.....	157
Ethernet.....	7
ethernet.....	20
Ethernet.....	157, 165
Ethernet host address EHA.....	158
exception.....	58
exclusive.....	162
executable file.....	144
executed.....	144
extensible.....	15
external linkage.....	100
file descriptor.....	91
file objects.....	15
FileID Table.....	168
FileIDs.....	166
FilesOpenBy.....	98
filler block.....	172
Flexible Intraconnect.....	157
fragmentation.....	117, 154
frame.....	157

Frame Check Sequence.....	162
free read.....	92
free write.....	92
FreeDiskBlock.....	168
frozen.....	92
Frozen.....	96
functional decomposition.....	130
functional design.....	11
functionality.....	136
gateway.....	126
gateway monitoring.....	133
GenerateUNO.....	176
generic.....	28
Generic Computing Element GCE.....	164
Generic Computing Elements GCE.....	6
global.....	18
global performance.....	4
global symbolic name space.....	99
goodaddresshint.....	44
hashing.....	158
HEREIAM.....	135
heterogeneous.....	5, 45
hierarchical.....	18
hierarchically structured.....	99
high-bandwidth.....	157
hint.....	13
Hints.....	44
host dependent role designator.....	51
host monitoring.....	136
host probe.....	132
HostAddress.....	177
host-dependent.....	151
hosthumber.....	27
hostincarnation.....	27
HostIncarnation.....	177
IgnoreMAddress.....	163
index.....	171
inherit.....	23, 55
initial directory.....	100
initial process load.....	146
initialization.....	134
InitScan.....	108
integers.....	61
integration.....	17, 46
integration cost.....	175
integrity.....	4, 14
Intel Corp.....	157
interact.....	126

interactive section.....	131
internal structure.....	14
Internet.....	5, 126
internet address.....	154, 177
Internet addresses.....	7
internet datagram.....	165
internet datagrams.....	153
internet gateways.....	152
internet header.....	154
internet protocol.....	20
Internet Protocol IP.....	9, 151
interprocess.....	33
interprocess communication.....	22, 29
Interprocess Communication IPC.....	5
interprocess communication IPC.....	13, 58
interprocess communication/coordination.....	164
Interrupt.....	45
invisible.....	127
invocation.....	38
invoke.....	33
invokemode.....	42
invokeonhost.....	31, 42
InvokeOnHost.....	58
I/O devices.....	123
IPC facility.....	179
IPCEnabled.....	49
ipreceive.....	39
ipsend.....	39
jam.....	162
kernel.....	12
key.....	47
key-value.....	63
key-value pair.....	15
key-value pairs.....	48
kill.....	48
labelled arc.....	99
Language Integration.....	60
LargeStepper.....	175
layers.....	151
Line printer.....	123
line printer managerz.....	124
link.....	100, 104
link target.....	100, 104
lists.....	61
load.....	52
load image.....	179
Load_Program.....	53
local address.....	154

local address field.....	155
local area network.....	5, 7
local network.....	126, 157
local networks.....	151
locate.....	31
Locate.....	47
logical name.....	27, 28
login.....	139
lookup.....	101
M68000.....	148
mainframe.....	6
Mapping Update.....	160
memory allocation.....	164
message.....	58
message oriented.....	33
message service.....	39
Message Structure Facility.....	179
message structure facility MSF.....	15
Message Structure Facility MSF.....	58
Message Structure Library MSL.....	58
messages.....	39
migratory file.....	90
migratory objects.....	14
missing blocks.....	172
monitoring.....	19, 129
monitoring and control station MCS.....	129
Monitoring and Control System MCS.....	129
Multibus.....	8, 148
multicast.....	154, 156, 178
multicast addresses.....	163
multicast-host address MHA.....	158
Multicast_Registered.....	161
multiple process.....	164
multi-window.....	132
mutual exclusion.....	178
MyAGS.....	49
MyUID.....	49
MyVLNAddress.....	163
name space.....	13, 99, 100
name tree.....	101
nametotype.....	27
network monitoring.....	129
network number.....	154
network traffic.....	134
new users.....	138
NextBlock pointer.....	168
node.....	99
non-migratory objects.....	13

non-terminal node.....	99
non-volatile.....	176
Normal file.....	168
Null values.....	63
object managers.....	12, 24
object model.....	22
object orientation.....	22
object types.....	12
object typing.....	12
octet.....	39, 67
octet ordering.....	158
octets.....	63, 154
open.....	91
Open.....	96
open.....	167
open operating system.....	164
OpenStatusOf.....	98
operating system.....	3
operation.....	31
operation switch.....	13, 22
operations.....	22, 58
operator's console.....	129
optional key.....	48
OrderOfUNOs.....	177
OriginOfUNO.....	176, 177
overflow blocks.....	171
packet size.....	36
pad octets.....	64
parent.....	55
partial name.....	100
partial symbolic name.....	104
pattern.....	101
peer-to-peer.....	39
performance.....	136
permanently bound.....	13
phases.....	136
physical local network.....	20, 152
PLN.....	157
polled messaged.....	131
Portability.....	59
portable.....	164
PPM.....	50
primal file.....	13, 16, 90
Primal File Manager.....	46
Primal File UID Table.....	91
primal process.....	33, 46
Primal Process Manager.....	46, 50
primal processes.....	17

primary symbolic access path.....	119
principals.....	19
Proceed.....	53
process.....	45
process control.....	45
process descriptor.....	48
process environment.....	139
process objects.....	16
Process Support Library PSL.....	179
Process_List.....	51
program carrier.....	17, 46
Program Carrier Manager.....	46
Program Carrier Manager Operations.....	55
program image.....	145
program support library.....	19
Program_Load_File.....	55
Program_Name.....	54
Program_Version.....	55
Pronet.....	157
protocol hierarchy.....	151
PurgeMAddresses.....	163
random read.....	174
random write.....	174
Read.....	92, 96, 97
ReadACL.....	97
ReadDescriptor.....	97
ReadDirectory.....	110
ReadEFSFileBlock.....	167, 173
ReadEntry.....	108
reader-writer.....	91
ReadRandomEFSFileDataBlock.....	174
ReadWrite.....	92, 96
ReadyToStart.....	54
real-time.....	131
reassembly.....	154
receive.....	33
Receive.....	47, 58
ReceiveVLNDatagram.....	163
reconfiguration.....	131
records.....	61
recovery.....	93
register.....	163
relative name.....	100
relative symbolic name.....	140
reliability.....	118, 136, 175
reliable delivery.....	156
Remove.....	107
RemoveFromACL.....	98

replicated objects.....	14
replication.....	117
reply.....	31, 58
Report_Process_Descriptor.....	47, 49
Report_State.....	53
request-reply paradigm.....	31
required features.....	22
required keys.....	48
ResetVLNInterface.....	162
resident.....	135
resource management.....	4
resource-sharing.....	11
Resource_Test.....	55
restart.....	134
restarts.....	176
Resume.....	45
RetainWrites.....	96
revision.....	101
role designator.....	51
ROM.....	135
root.....	99
root directory.....	101
root portion.....	114
Running.....	54
S16I.....	67
S32I.....	67
salvager.....	172
Scalability.....	4
ScanDirectory.....	108
Search_All_Descriptors.....	55, 56
secondary symbolic access path.....	119
self-describing.....	60
SendToHost.....	47, 58
sendtohot.....	33
SendVLNDatagram.....	163
sequence.....	154
sequencenumber.....	27
SequenceNumber.....	177
Sequencing guarantees.....	157
sequencing property.....	156
sequential.....	166
serializable.....	92
service.....	27
service monitor.....	130
service probe.....	133
service probes.....	133, 136
Service_List.....	51
Session.....	54

session.....	138
session agent.....	126
session controller.....	138
session identifier.....	139
Set Access Rights.....	45
Set Priority.....	45
Short file.....	168
sink.....	72
site-based decomposition.....	130
SmallStepper.....	175
source.....	72
specific names.....	28
specific uid name.....	28
spooler.....	124
Standard External Representation SER.....	58
standard types.....	61
startup.....	176
Status_Probe.....	51
Stop.....	53
stream.....	54
streams.....	56
structural design.....	11
structured objects.....	14
substitutability.....	4, 20
Substitutability.....	151
substitutability.....	157
substrate.....	5
subtype-supertype.....	24
Survivability.....	4
Suspend.....	45, 53
Suspended.....	54
symbolic catalog.....	29
symbolic links.....	104
symbolic name.....	25
symbolic name space.....	99
symbolic names.....	18
synchronization.....	91
system clock.....	164
system decomposition.....	12
table-driven.....	36
tape drive.....	123
tape drive manager.....	124
TCP.....	72
TCP stream.....	165
Telnet.....	6, 165
terminal access computers TAC.....	126
terminal concentrator.....	6
terminal device.....	138

Terminate.....	45
thawed.....	92
Thawed.....	96
transaction identifiers.....	175
transation protocol.....	15
Transmission Control Protocol.....	151
Transmission Control Protocol TCP.....	9
transparent.....	127
transport.....	152
trap logging.....	132
traps.....	131
type tag.....	62
TypeOfAccess.....	96
TypeOfSynchronization.....	96
typetaname.....	27
U161.....	67
U32I.....	67
uid.....	25, 30
UID.....	67
uid naming.....	22
uid table.....	25
unattended.....	54
uniform.....	22, 127
uniformity.....	3, 11
unique identifier.....	25
unique identifier UID.....	13
unique numbers.....	175
UNIX.....	72, 148
UNO.....	175
UNO generator.....	178
UNO size.....	176
user identity.....	16
user interface.....	12, 19, 126
user session.....	54
User_Environment.....	55
users.....	19
Utility hosts.....	8
VAX 11/750.....	148
version.....	101
Virtual Local Net.....	7
virtual local network.....	20, 31
Virtual Local Network VLN.....	151
VLN.....	7
VMS.....	148
VPMap.....	158
well-known.....	177
well-known UNOs.....	177
working directory.....	100

workstation.....	126
workstations.....	6
Write.....	97
WriteDescriptor.....	97
WriteDirectory.....	111
WriteEFSFileBlock.....	167, 173
WriteRandomEFSFileBlock.....	174
Xerox Corp.....	157
XMD.....	164

Report 5261

PART B

Cronus, A Distributed Operating System:
Interim Technical Report No. 2

R. Schantz, B. Woznick, G. Bono, E. Burke, S. Geyer
M. Hoffman, W. MacGregor, R. Sands, R. Thomas and S. Toner

Prepared for:

Rome Air Development Center
Griffiss Air Force Base

Table of Contents

	B-
1 Introduction.....	1
2 GCE Network Software.....	2
3 GCE Disk Software.....	7
4 C/70 Network Facilities.....	8
5 VAX/VMS Network Software.....	9
6 Cronus Network Performance.....	10
6.1 Introduction.....	10
6.2 Explanation of Terms.....	10
6.3 Results.....	13
6.3.1 Go-Bit-Test.....	13
6.3.2 Datagram Copy Test.....	14
6.3.3 Raw Ethernet Back-To-Back Test.....	15
6.3.4 Raw Ethernet Round Trip Test.....	16
6.3.5 IP Back-To-Back Test.....	17
6.3.6 IP Round Trip Test.....	18
6.4 Discussion.....	18
6.4.1 Performance Enhancements Suggested By The Tests.....	18
7 Message Structures.....	24
7.1 Purpose and Scope.....	24
7.2 Design Issues.....	25
7.2.1 Objectives.....	25
7.2.2 A Taxonomy of Message Structure Conventions.....	27
7.2.3 Four Existing Conventions.....	30
7.2.3.1 NSWB8.....	30
7.2.3.2 The Internet Message Protocol.....	31
7.2.3.3 The NBS Message Format.....	33
7.2.3.4 Courier.....	35
8 Cronus System Libraries.....	38
9 Configuration Management.....	39
10 Standards, Policies, and Procedures.....	40
11 System Documentation.....	41
11.1 User Manual.....	41
11.2 Operations.....	41
11.3 Program Maintenance Manual.....	42

1 Introduction

This part of Interim Technical Report #2 consists of a series of short notes and reports of activities performed during the period. Principal among these are discussions of the various activities supporting the development of the system, and of the progress on the components of the system support environment: gce, network, C70 constituent operating system modifications.

The following accomplishments during the preceding period are described in this section:

- o completed the integration of the Ethernet local area network into the GCE, the VAX/VMS and the C/70 UNIX hosts
- o completed the integration of IP and TCP protocols into the GCE, VAX/VMS and the C/70 UNIX and interfaced this software to the Ethernet layers using the Virtual Local Network concepts
- o completed a CMOS-based Telnet program to support interactive access to other cluster hosts from the GCE
- o completed the integration of a disk subsystem into the GCE CMOS System
- o completed the design and part of the implementation for a elementary file system for the GCE, which is to serve as the base implementation for the Cronus file system.
- o completed a set of performance tests to evaluate the Ethernet hardware and software, as well as IP and TCP implementations
- o developed and installed a system configuration management plan for source code and documentation
- o developed code for and assembled library functions needed to support the development of Cronus system components
- o established standards and approaches to achieve the high degree of program portability required by our system implementation approach

2 GCE Network Software

A Generic Computing Element (GCE) is a multi-purpose microcomputer system which is custom configured for a variety of special purpose host roles in the Cronus architecture. It is based on M68000 processor technology in a Multibus chassis. The operating system for the GCE hosts is named CMOS.

Work has been completed in all the following areas of network protocols on the GCE under CMOS.

- 1) Ethernet Protocols
- 2) Virtual Local Net
- 3) Internet Protocols
- 4) TCP
- 5) Telnet

The objective of this work is to implement the communication protocols as a base for the Cronus communication and control systems.

The initial plan for bringing up network software on the GCEs was to adapt the BBN-UNIX version. Subsequently, it was brought to our attention that there was, at MITRE, an IP - TCP implementation based on an older version of the CMOS operating system. Although the BBN-UNIX IP - TCP was more complete (the IP layer supported ICMP and UDP), the MITRE implementation was used because it would be easier to bring up on the current version of CMOS.

The MITRE IP -TCP was encapsulated, to minimize the number of changes necessary to run on the new version of CMOS. The encapsulation was soon completed and was followed by the integration with the Ethernet software. A TCP test program was written, and the debugging phase.

Figures 1 and 2 indicate the test configurations used to demonstrate IP and TCP. Test 1 demonstrated character-at-a-time functionality between two GCEs on the Ethernet using TCP protocol. Test 2 showed the same functionality Between a GCE and the C70 UNIX (using its ARPANET 1822 interface) with the Cronus gateway interposed. In order to get IP to work with the Cronus gateway, the VLN address mapping scheme was added to the local network code. After this was done, test 2 was performed

successfully. Test 3 demonstrates the functionality of Test 2 but directly through the Ethernet.

Figure 1
TCP FUNCTIONAL DEMONSTRATIONS

Test 1:



Test 2:

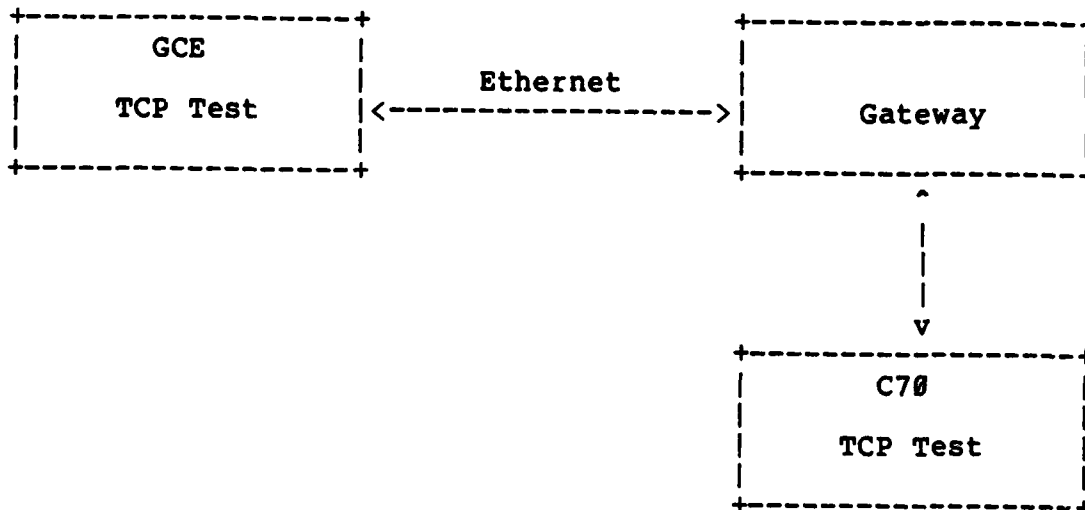
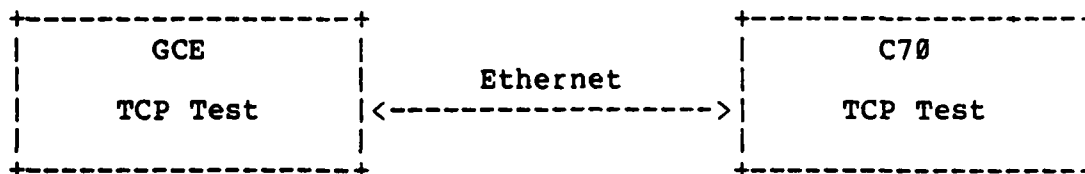


Figure 2

TCP FUNCTIONAL DEMONSTRATIONS (Continued)

Test 3:



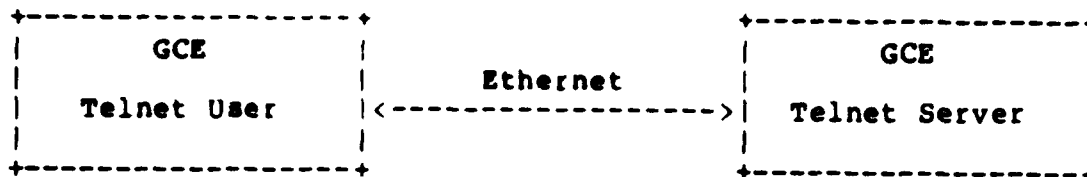
Tests 1, 2 and 3 have all been successfully completed.

Finally, we developed a simplified GCE user Telnet program. This is a prototype implementation that is a modification and extension of the TCP test program. In the course of debugging Telnet, a number of bugs were found and fixed in the Cronus gateway. The GCE Telnet has been used to login to Arpanet hosts from a Cronus network GCE, to perform various commands on these hosts (such as listing directories), to logout, to disconnect, to re-connect, and to login again. The following are the Telnet test configurations.

Figure 3

TELNET FUNCTIONAL DEMONSTRATIONS

Test 1:



Test 2:

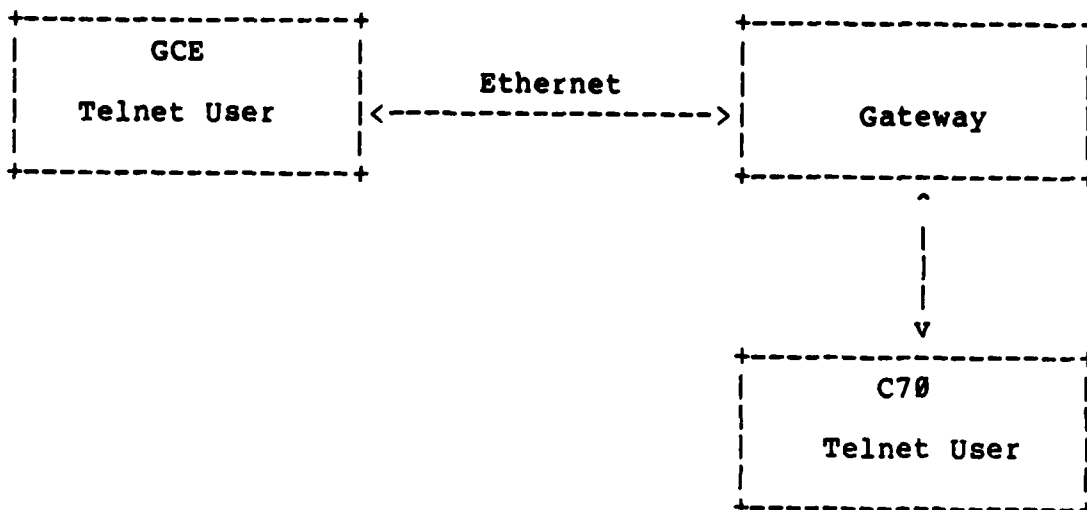
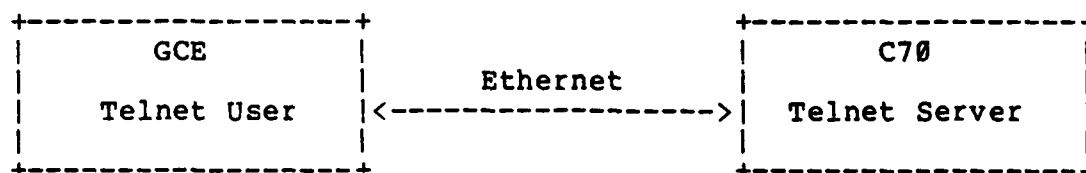


Figure 4

TELNET FUNCTIONAL DEMONSTRATIONS (Continued)

Test 3:



Test 1 was a trivial Telnet to Telnet connection between GCEs. Test 2 showed the GCE capability for logging in to a remote host through the Cronus gateway. Test 3 demonstrates the compatibility for the C70 of the implementation with other host types over the Ethernet.

3 GCE Disk Software

The Interphase SMD 2181 Storage Module Controller was chosen as the disk controller for the Cronus GCEs. Production problems delayed delivery of this controller, so we initially used an SMD 2180 disk controller (with which the SMD 2181 was advertised to be upward-compatible) and began implementing the elementary File System. We constructed a CMOS device driver, Elementary File System Initialization program and the CreateEFSFile, DeleteEFSFile, and sequential read and write routines for the SMD 2180. (See Section 14 of Part B for the description of the Elementary File System).

When the SMD 2181 controller arrived, we found that it was not completely compatible with the SMD 2180, so we converted our code to run on the new controller. There were a number of other problems we encountered in integrating the new controller, causing extremely high error rates. After trying new firmware for the controller, which did not fix the problem, we shipped one of our controllers and disk drives back to Interphase, so that they could find and fix the problem, which turned out to be a byte order incompatibility in accessing Multibus memory.

Another problem occurred in which some random bytes would at times be overwritten with the wrong data when reading a block of data from the disk. This problem was traced to bus contention occurring when the controller transferred the data from its on-board buffer into its Multibus memory. Interphase provided us with a patch to correct this, which we have installed. The board now seems to run with very few problems, although some advertised features of the board (such as error correction) are not implemented in the current version of the microcode.

The GCE implementation of the Elementary File System, which is now largely implemented, is being tested using the SMD 2181 controller.

4 C/70 Network Facilities

The task to connect the C/70 UNIX to the Cronus Ethernet is broken up into three pieces. The first piece is building hardware interface (called MIENI) to connect the C/70 to the Interlan NM10 Ethernet controller. The second is writing and debugging C/70 microcode to drive the hardware. Last is the UNIX driver connecting hardware to the UNIX operating system.

The hardware was debugged in the fall of 1982. Since that time, two bugs were found while debugging the software. The first bug was a bad interaction between the NM10 and the MIENI board which caused the microcode to read the status byte twice instead of once. The second bug caused the interface to stop transmitting when it was highly loaded. There was a timing problem in the MIENI causing it to occasionally send a bad command.

During this time, the second MIENI board was brought up and is now operational. We also received new NM10 boards from Interlan which fixed some minor problems in the hardware.

The microcode has been stable for four months. While debugging the C driver, one bug was discovered. During some interrupts, the microcode neglected to change the memory maps to be in UNIX kernel space, which caused the microcode to fetch data from the wrong address.

The UNIX driver is still being debugged. At this moment one known bug remains that causes the network software to add random data to the good data. It is being attacked with the help of the network software group. We expect this bug to soon be found and corrected.

5 VAX/VMS Network Software

IP and TCP for the VAX/VMS have recently been installed in our test configuration. After dealing with a number of operational difficulties and deficiencies in handling Ethernet address translations, we now have the software running. We have added the Virtual Local network software to the 3Com Ethernet device driver, and are now in the process of testing and evaluating the IP and TCP implementations.

6 Cronus Network Performance

6.1 Introduction

Over the past several months, performance measurements have been made on the Ethernet local area network. These tests were designed to give us concrete information on the performance limits of the Ethernet and of IP.

The three most important characteristics of the Cronus network are throughput, delay, and reliability. Throughput is the rate at which data can be sent from one host to another. Delay is the time from initiation of transmission of a single datagram, until the datagram is available to the receiving process. Reliability is measured by determining the percentage of datagrams which do not reach the receiving process; datagrams are lost because of hardware errors or because the receiver can't keep up (overspeeding). These three characteristics are a function of the size of datagrams. Other interesting dependencies include the Ethernet addressing mode, and the buffering and synchronization techniques.

The tests measure the characteristics between two MC68000-based GCEs on an otherwise unloaded network. The reported results are the average values for 2000 datagrams. The measurement programs run under the CMOS operating system, and use standard CMOS i/o and synchronization techniques.

6.2 Explanation of Terms

1) Datagram size

Datagram size is the number of octets in the data field of a datagram. It does not include header or trailer information used by various protocols. For example, an Ethernet datagram (referred to as a 'frame' by the standard) consists of a 14 octet header which includes addresses and a type field, 46 to 1500 octets of data, and a 4 octet CRC. The smallest Ethernet datagram is 64 octets, and the largest is 1518 octets including protocol overhead. Since datagram size refers to the data field only, for the Ethernet it may range from 46 octets to 1500 octets. IP datagrams add another 20 overhead octets, and may range in size from 20 octets (no data) to the maximum allowed by the underlying protocols, which in the case of Ethernet is 1500 octets. Subtracting the header size, the datagram size for IP may range from 1 octet to 1480 octets. Note that IP datagrams with a datagram size of 26 octets or less are transparently padded when encapsulated in an Ethernet

datagram, because minimum Ethernet datagram size is 46 octets. This definition of datagram size is chosen as the throughput or delay measurement depends on the amount of data sent.

2) Throughput

Throughput, is the rate at which data can be transferred between hosts by sending datagrams back to back as quickly as possible. If the transmitting host can send data faster than the receiving host can accept it, a condition known as overspeeding results. Datagrams will be lost if the receiver is not ready to accept them. Throughput is measured in bits per second (Mbps = megabits/second, Kbps = kilobits/second) and datagrams per second.

3) Bandwidth Utilization

Assuming that the Ethernet's raw bandwidth is 10.0 Mbps, bandwidth utilization value is the percentage of the raw bandwidth used in transmission. The usable bandwidth is less than 10.0 Mbps, if one takes into account the overhead of protocol headers. This usable bandwidth varies according to the datagram size; it seemed a dubious complication to compute the percentage of usable bandwidth utilization.

4) Errors

All datagrams for which errors are detected in either the Ethernet or IP layers are discarded. Statistics are kept by the Ethernet driver and the IP protocol handler, and are accessible to the test programs. Ethernet transmission errors that are detected by the controller or driver include 1) too many collisions encountered in attempting transmission, and 2) controller errors. Ethernet receive errors include 1) CRC errors, 2) framing errors, 3) controller errors, and 4) no receive request outstanding for the data link type of the received datagram. The controller also detects the lack of receive buffer space on the controllers but this error is not counted because these datagrams are discarded with no indication to the driver. Lost datagrams not accounted for in other statistics are assumed to have been discarded due to lack of controller buffers.

IP errors which are detected include 1) datagrams received on the Ethernet but not addressed to the host, 2) lack of IP buffer space, 3) incompatible version number, 4) datagram received with a protocol field for which there is no open stream, and bad IP checksum.

Of all these errors which can be detected, the only error ever detected was the Ethernet 'no receive request outstanding for data link type'. This is not an error of the Ethernet itself, but is a consequence of the decision to have the CMOS Ethernet device driver discard datagrams for which no outstanding receives have been queued. Other strategies might have included buffering of datagrams in the driver itself, waiting for receives to be queued. This approach was not taken because higher-level protocols are better equipped to make buffering decisions. No datagrams were discarded for any other reason, including lack of controller buffers.

5) Received Datagrams

A datagram is counted as received when it reaches the destination with no errors detected.

6) Dropped Datagrams

A datagram is counted as dropped if it is received properly by the hardware, but discarded because there was no receive request outstanding for the data link type.

7) Overspeeding

If the receiver cannot keep up with the transmission rate, a condition known as overspeeding occurs. Though buffering can absorb bursts of datagrams coming in at high speeds, it cannot prevent overspeeding if the transmitter consistently transmits datagrams faster than the receiver can accept them. Buffering occurs in the controller, with room for two datagrams, and in the user process. User buffers are queued to receive datagrams of a certain data link type, are returned to the receiver process when received datagrams have been placed in them. Overspeeding occurs if either of these two buffer resources is exhausted. IP adds an extra level of buffering, with IP buffers being passed down to the Ethernet layer to be used as Ethernet buffers. Overspeeding can also occur if the pool of IP buffers is used up. During the course of the tests, the only type of overspeeding observed was caused by Ethernet user buffers not being queued fast enough.

8) Delay

The round trip delay is the time to send a datagram of a certain size and to receive a datagram of the same size as an acknowledgement. The one-way delay is one-half of the round trip delay. Delay is measured in milliseconds/datagram.

9) Go Bit Test

In the go-bit test, the controller repeatedly transmits the same datagram, with no copying of data. When a transmit complete interrupt occurs, the program restarts the transmission. This measures the speed of the controller hardware. The test does not implement the portions of the Ethernet Data Link protocol implemented in software for the 3Com Controller

10) Datagram Copy Test

This test is run as a stand-alone program without CMOS or the CMOS Ethernet device driver. A maximum size datagram is copied into the controller's transmit buffer and transmitted. When the controller interrupts, signalling completion of the transmit, the datagram is copied again and the operation repeats. This test measures the overhead that CMOS and the Ethernet device driver introduce. As with the go-bit test, the software portions of the Ethernet Data Link Protocol are not implemented.

11) Back To Back Test

The back to back test measures the speed at which a GCE can transmit data with no acknowledgements. The transmitting machine sits in a loop, sending datagrams as quickly as possible, and the receiver tries to keep up. This test was run for both raw Ethernet, and IP.

12) Round Trip Test.

The round trip test measures the speed at which a pair of GCEs can transfer data with acknowledgements. The acknowledgement datagrams are the same size as the message datagrams to permit the calculation of one way delay. The test does not implement a reliable channel with sequence numbers, timeouts, and the like.

6.3 Results

6.3.1 Go-Bit-Test

The Go-Bit Test indicated that the 3Com Ethernet controller meets the Ethernet specifications. It helps verify the timing of the other tests, since the results were predictable from the specification.

The clock rate on this Ethernet is 10.0 Mhz, which means that one bit-time is 100 nsec. No controller can sustain a data rate of 10 Mbps, since the protocol requires a 9.6 microsecond interdatagram spacing and a 64-bit preamble for hardware synchronization. The maximum size of an Ethernet datagram not including header and trailer, is 1500 octets, or 12000 bits. Each datagram also includes 144 bits of header and trailer, and requires the 64 bit preamble, making the number of bit times per datagram 12208. In addition, 96 bit times are required for the interdatagram spacing delay, making the total number of bit times to transmit a single, maximum size datagram 12304. The maximum usable bandwidth of the Ethernet is thus

$$10 \text{ Mbps} * 12000 / 12304 = 9.75 \text{ Mbps.}$$

The measured maximum usable bandwidth is 9.6 Mbps, close enough to be confident that the hardware is functioning normally, and that the timing of the performance measurements is accurate. The slight reduction in usable bandwidth is attributable to the interrupt latency of the MC68000, execution time of the interrupt handler, and the occasional preemption of the interrupt handler by a memory refresh interrupt.

6.3.2 Datagram Copy Test

Any real data transfer using the Ethernet must encapsulate the data to be transmitted, and then copy the data into the controller's buffer. The CMOS Ethernet device driver performs these operations, as well as handling the portions of the data link protocol not handled in hardware. A stand-alone program which copies a datagram to the controller's buffer before transmitting it provides a useful point of comparison to the Ethernet device driver, since such a program would give an upper bound on performance of simple data transfer. The maximum throughput in the Datagram Copy Test was 3.2 Mbps for 1500 octets of data per datagram. This is nearly 1/3 of the bandwidth of the network.

In performing other tests, it was found that datagrams with an odd number of octets require substantially more time to process than do datagrams with an even number of octets. For example, the throughput for the Datagram Copy Test for datagrams with 1499 octets of data was 2.06 Mbps, only 64% of the throughput for datagrams with 1500 octets of data. The difference was caused by the data movement routine, and is the result of interactions between the architecture of the MC68000 and the 3Com Multibus Ethernet Controller. The cpu must copy the

Ethernet datagram from its local memory into the controller's buffer in multibus memory space. To save complexity in the controller, the datagram is aligned with the end of the buffer, rather than the beginning. The controller is given an offset to the start of the datagram, and stops transmitting when it gets to the end of the buffer. This eliminates the need for a counter in the controller; however, it does introduce the odd-even disparity. The M68000 has a 16-bit external data bus, which means that a copy algorithm can move the data in 16-bit chunks, but the data transfer must be aligned on an even-byte for both address source and destination. It is thus impossible to use 16-bit move instructions to transfer a data block from an even-aligned address to an odd aligned address, or vice-versa. This case occurs when a datagram has an odd number of octets in it, since the start will be an odd number of bytes before the end of the even-aligned controller buffer. In the even-size case, data is copied using the more efficient word-sized instructions, but must be copied using byte moves for odd-sized datagrams.

6.3.3 Raw Ethernet Back-To-Back Test

The Ethernet back-to-back test was run over a range of datagram sizes from 50 to 1500 octets, with separate tests for specific, broadcast, and multicast addressing modes. The differences in throughput modes and delay for different addressing modes were negligible. Even for multicast addressing, which uses a lookup table in the driver to perform address recognition, an increase in processing time was seen. The time spent copying the datagram to the controller's buffer and in process synchronization is a high percentage of the overall time spent in the Ethernet driver, and small changes in the rest of the processing are thus insignificant. The results given here are for the specific addressing mode.

Raw Ethernet throughput ranged from 90 Kbps for 50 octet datagrams, up to 1.39 Mbps for 1500 octet datagrams (see Figure 5). A single GCE can utilize about 14% of the available bandwidth of the network under idealized circumstances. The relationship between datagram size and throughput is fairly smooth, with some bumps in the curve for small datagrams. The curve inexplicably takes a small dip between 1300 and 1350 octets. The datagram sizes ranged from 50 octets to 1500 octets by 50, so all of the test points are for even-sized datagrams. Datagram throughput ranged from 227.0 datagrams/second for 50 octet datagrams, to 115.7 datagrams/second for the maximum-sized 1500 octet datagrams (see Figure 6). The shape of the datagram throughput curve for small datagrams is difficult to explain. It

is probably related to the slight non-linearity through the same range of datagram sizes seen in the throughput curve.

Buffering and synchronization decisions in the receiver played a substantial role in determining whether it could keep up with the transmitter, avoiding overspeeding. The simplest scheme uses a single receive buffer and one outstanding receive request. The receiver process enters a wait state after resubmitting the buffer and request. Receiving a datagram thus entails the following: 1) copying the datagram to the supplied buffer from the controller, 2) signalling the waiting receiver process, 3) rescheduling of the receiver process, 4) the switching into the process, and the 5) resubmitting of the buffer and receive request. There is queue management overhead for signalling and scheduling, as well as in the scheduler overhead and the context switching time. This simple technique resulted in severe overspeeding for small datagrams, because the fixed overhead of CMOS synchronization is a relatively high percentage of the total overhead. (See Figure 7.)

A somewhat more sophisticated method uses multiple receive buffers and outstanding receive requests. When several datagrams come in back to back, the device driver is able to buffer them at interrupt level if several receive requests are outstanding. Each buffered datagram causes a signal to be queued for the receiver process. When the receiver process is rescheduled, it is able to handle a number of completed receives without the overhead of CMOS rescheduling and context switching for each received datagram. This scheme was much more successful than the simple scheme described above. With only two buffers, the receiver was able to keep up with the transmitter completely, resulting in no dropped datagrams (see figure 3).

6.3.4 Raw Ethernet Round Trip Test

The Ethernet Round Trip Test was run over the same range of datagram sizes as the Back-To-Back Test; 50 octets through 1500 octets. The transmitter sends a datagram and waits to receive an acknowledgment datagram from the receiver. The receiver, after receiving a datagram, sends an acknowledgement back which is the same size as the test datagram. Since both the receiver and transmitter use the same techniques for transmitting and receiving datagrams, and the datagrams going in both directions are the same size, dividing the round trip time by two gives the time from starting transmission until the datagram is available to the receiving process. This time is the one-way delay. The one-way delay ranged from 7.38 milliseconds/datagram for a

datagram size of 50 octets, through 13.28 milliseconds/datagram for a datagram size of 1500 octets. The relationship between datagram size and delay is exactly linear (see Figure 8).

6.3.5 IP Back-To-Back Test

The IP Back-To-Back Test was run for datagram sizes ranging from 50 to 1450 octets, so as to be directly comparable to the Ethernet test. The largest IP datagram which can be encapsulated in an Ethernet datagram is 1480 octets. The throughput ranged from 40 Kbps for a datagram size of 50 octets, through 830 Kbps for a datagram size of 1450 octets (see Figure 5). Comparing these figures to those for raw Ethernet, IP throughput ranges from 44% to 60% of Ethernet throughput for the same datagram size. The IP throughput curve is smoother than the corresponding Ethernet curve. All datagram size test points are even numbers. Datagram throughput ranged from 89.0 datagrams/second for 50 octet datagrams, through 71.4 datagrams/second for 1450 octet datagrams. Unlike the Ethernet datagram throughput curve, the IP datagram throughput curve is smooth.

Though IP uses multiple receive buffers, the IP receiver program could not keep up with the transmitter, even for large datagrams. This suggests that the fixed overhead of the IP protocol and the CMOS synchronization, scheduling, and context switching outweighs the per octet overhead of data copying (see Figure 9). The increase in dropped datagrams from 50 octet datagrams through 450 octet datagrams, and the subsequent sharp reduction can be explained by examining the buffering algorithm. For datagrams up through 450 octets, IP allocates a buffer which is the exact size of the datagram, copies the data from the receive buffer into this buffer, and gives the buffer to the receiving process. The receive buffer is then reused. For larger datagrams, the receive buffer is passed to the receiving process with no data copying, and a new receive buffer is allocated to take its place. This algorithm is motivated by the desire not to waste a full-sized buffer on small datagrams. The extra overhead of the copy operation for small datagrams causes the number dropped datagrams to increase through a datagram size of 450 octets. The sharp drop shows where IP switches to the alternate buffer management scheme. Datagrams are dropped because there are no outstanding Ethernet receive requests when a datagram comes in. Because the multiple buffer scheme for raw Ethernet avoids some of the the fixed overhead of CMOS, it provides an increase in the basic receive rate sustainable by the program. No such reduction in overhead is achieved by the multiple buffering scheme in IP, so no improvement in the dropped

datagram rate is seen.

6.3.6 IP Round Trip Test

The one-way delay for IP datagrams ranges from 19.1 milliseconds/datagram for 50 octet datagrams, through 24.8 milliseconds/datagram for 450 octet datagrams, in which IP is using the copy buffer management algorithm, and then from 20.0 milliseconds/datagram for 500 octet datagrams through 23.1 milliseconds/datagram for 1450 octet datagrams, using the no copy buffer management algorithm (see Figure 8). Within these two ranges, the curve is linear.

6.4 Discussion

6.4.1 Performance Enhancements Suggested By The Tests

As was seen before, IP throughput is between 44% and 60% of raw Ethernet throughput for the same size datagrams. Encapsulating data in an IP datagram is not inherently more complex than encapsulating it in an Ethernet datagram. The performance penalty is mostly a result of not having tuned the implementation of IP. Upon examination of the test results, several areas for improvement suggest themselves.

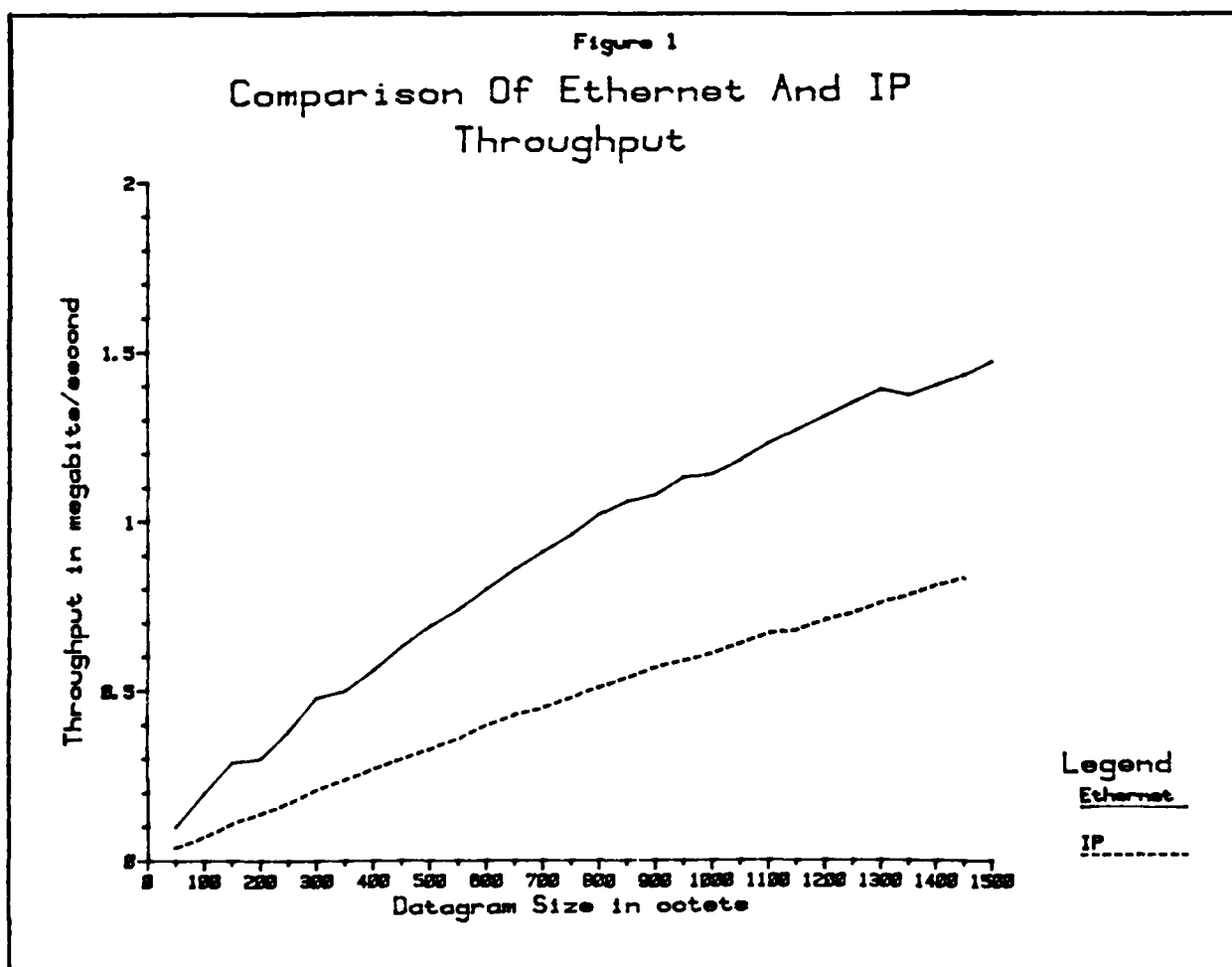
1) Encapsulation Of Odd-Sized Ip Datagrams

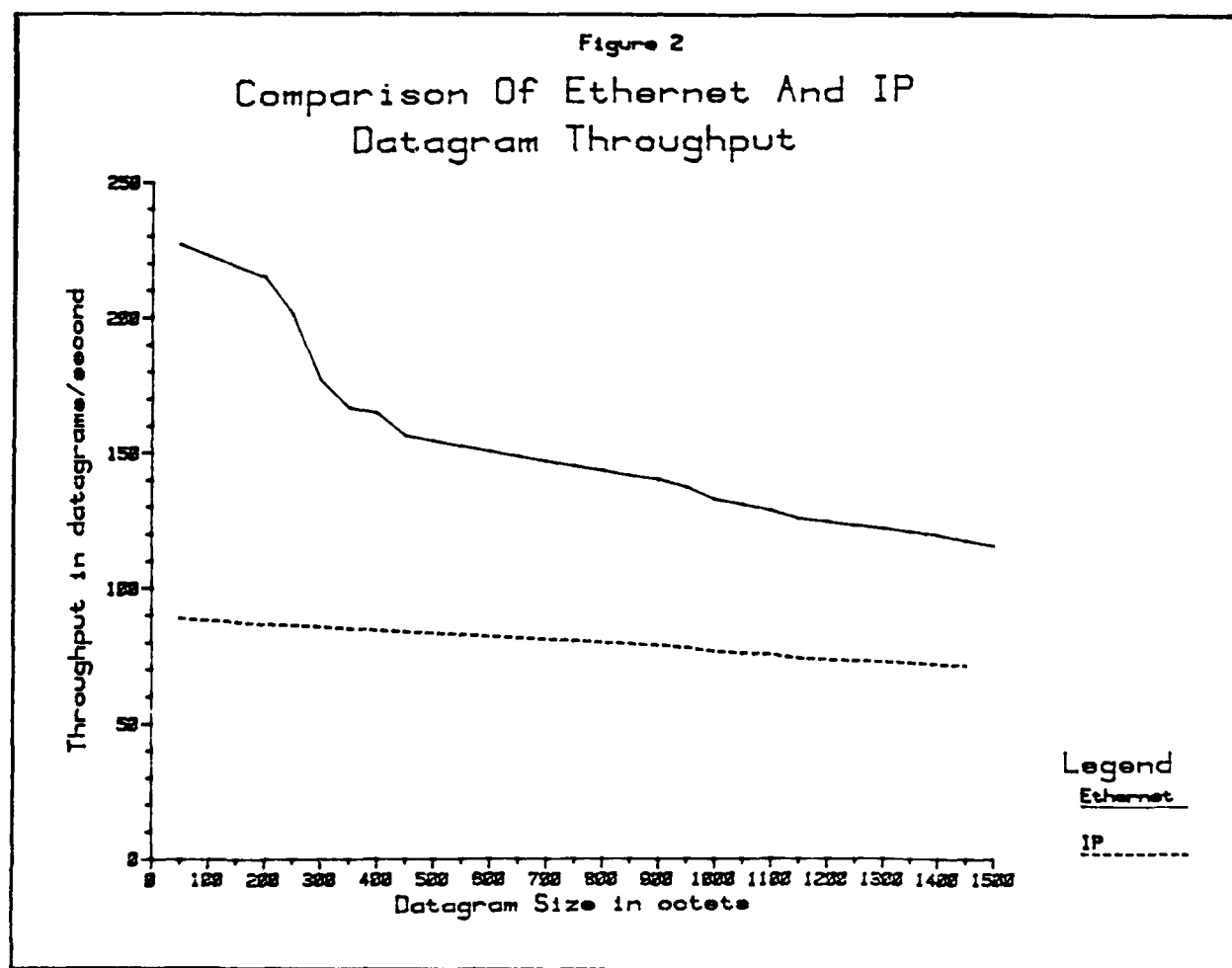
Because there is such a disparity in performance between odd- and even-sized datagrams that is inherent in the interaction between the controller and cpu, a substantial gain can be achieved for odd-sized datagrams by transparently padding them to an even length.

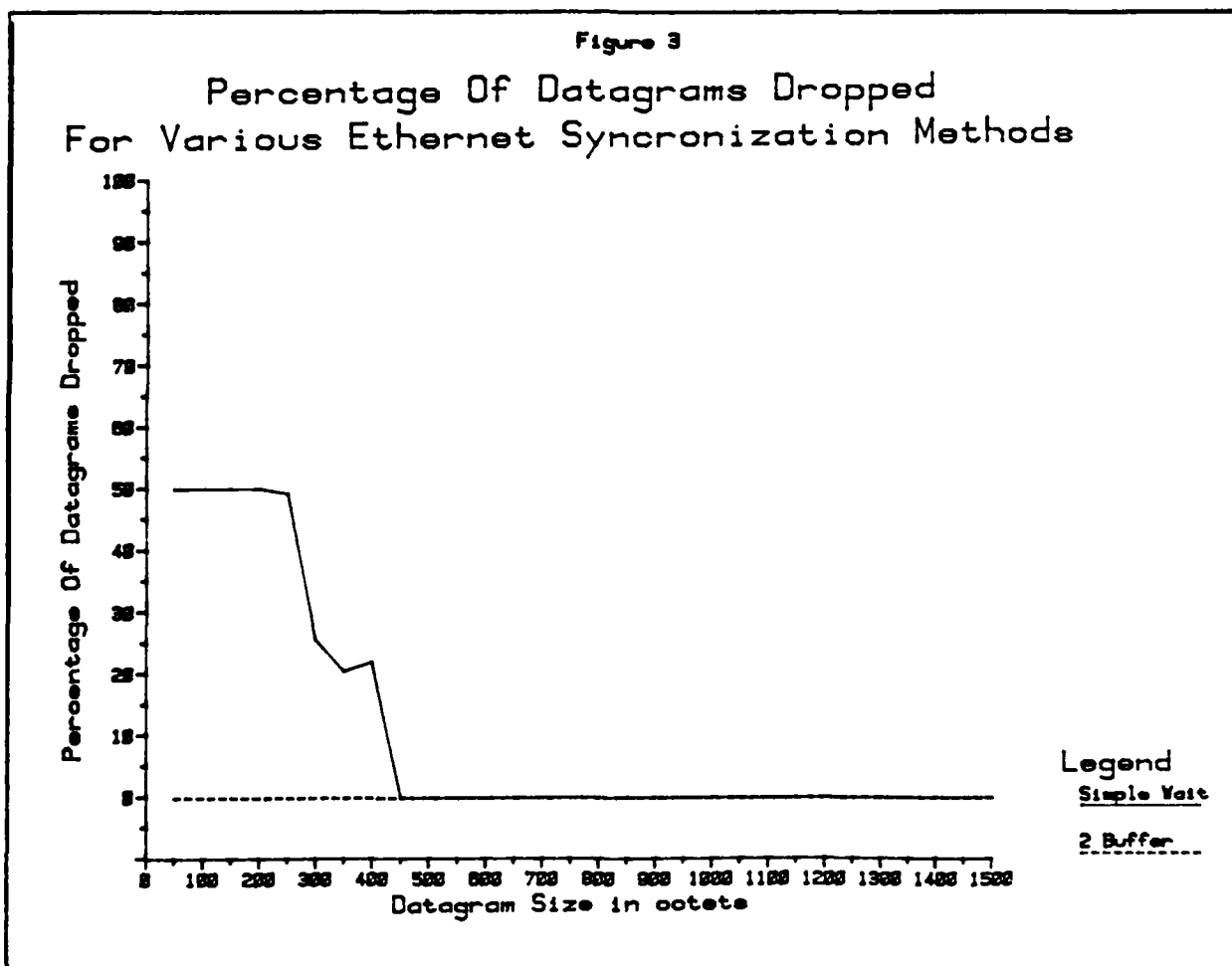
2) IP Buffer Management Algorithms And CMOS

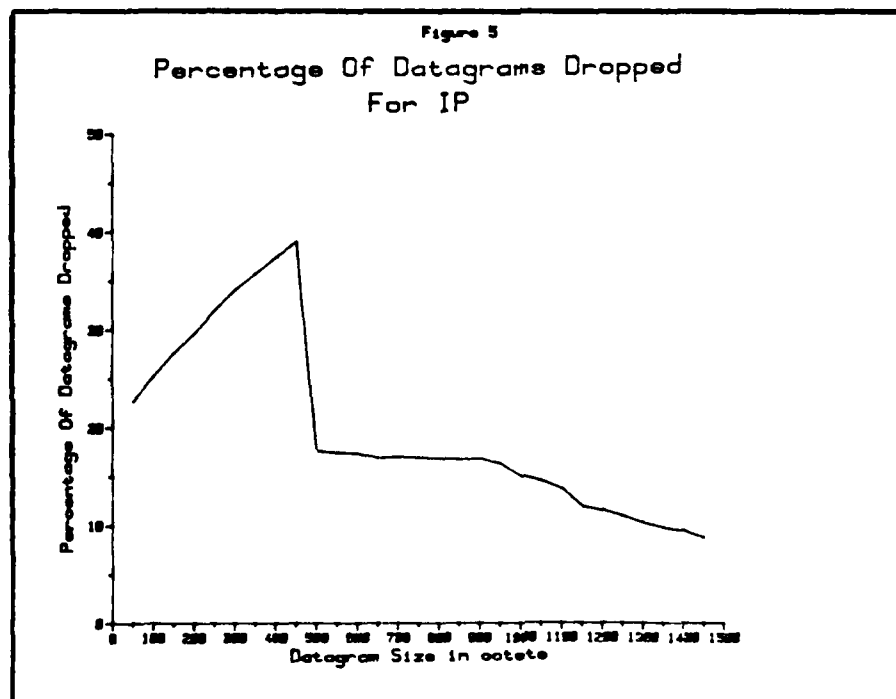
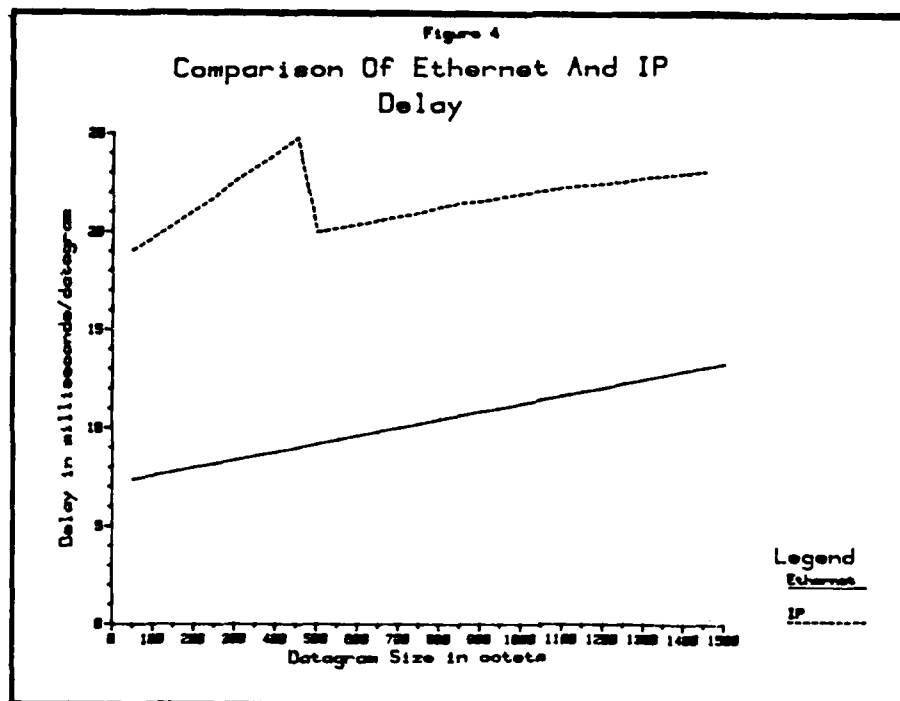
The current implementation of IP does multiple buffering of receives at the IP level, rather than at the Ethernet level. The process that handles Ethernet receives merely signals IP that a datagram has arrived, passing the buffer up. IP either copies the data to a small buffer and passes this buffer to the client process, or passes the receive buffer directly. It then queues a buffer indirectly by passing it to the Ethernet level, which then actually requeues the receive request. Separating the Ethernet and IP levels is probably a good idea,

since it allows easier implementation of the VLN. But CMOS process management overhead is high enough so that running different protocol levels as separate processes and buffering in a higher-level protocol causes an unacceptable dropped-datagram rate. These tests measure only pairwise communication on an unloaded network. In actual operation, if multiple transmitters address the same receiver simultaneously, burst receive data rates may far exceed the rates measured in these simple tests, causing even worse performance. It is important that resubmission of receive requests occur as quickly as possible. There are two possible solutions to this problem: either buffer incoming data as close as possible to the device driver and use a more sophisticated buffering algorithm, and/or find the bottlenecks in CMOS process management and eliminate them.









7 Message Structures

7.1 Purpose and Scope

The message structure selected for the Cronus system is described in Section 6 of Part B. The discussion in this section is an analysis of alternatives considered in making these decisions.

Communicating processes in a distributed operating system will not usually have access to shared memory or common state information at primary memory speeds, for two reasons: 1) two processes may be on different processors which do not share physical memory; 2) even when processes are on the same host and could interleave accesses to the host's primary memory, it may be impractical or undesirable to do so. Shared memory may be impractical because of its complex interactions with virtual memory support, for example; it may be undesirable because of the importance of the programming paradigm that allows processes to communicate only through messages.

Without system-wide conventions for message structure Cronus components would find it very difficult to talk to one another. Programs could make pair-wise agreements, but this approach becomes cumbersome and eventually unmanageable as the number of correspondents increases. More in keeping with the primary project objective of coherence and uniformity is a system-wide convention for message structure.

Conventions for message structure vary widely in their goals and scope of application. We assume here that the dominant goal is the regularization of control traffic in the Cronus system. Control traffic includes, among other things, requests for operations to be performed on objects, replies generated by operations, exception notices, and messages needed to coordinate distributed object managers. Control messages are usually short (tens to a few hundred octets) and are bounded by the maximum datagram size (a few hundred to a few thousand octets). Because control messages are often in the critical path to completion of an interactive command, performance is a major issue--messages should be compact, and efficiently composed and parsed.

Electronic mail an important example of structured messages which are not control messages. Conventions for the structure of mail messages meet very different goals than those for control traffic. Since the delivery of mail occurs as a background activity, relatively large space and time overheads can be tolerated. Mail messages are often large (tens of thousands of octets) and rarely smaller than a large control message (an

ARPANET mail message containing one sentence of text is about 300 octets long). Mail messages can be very highly structured, and fields such as "Date" and "Sender's Phone Number" that have immediate significance to human readers; thus there is a natural tendency to use English keywords as structural markers. Often the processes that send, receive, forward, or file mail need to interpret only a few fields in a header, passing or storing the remainder for human interpretation. Standardization of message structure is extremely important in electronic mail systems that extend through the internet and link mail programs, operating systems, and hosts provided by diverse vendors. Two mail standards are surveyed below. Finally, because mail messages are mostly text, they can be composed, altered, and examined by word processing tools such as text editors and formatters. The maximum benefit from electronic mail is obtained in systems which smoothly integrate mail and word processing tools.

Standards for the structure of electronic mail messages will be applied to mail in Cronus, but it should not be surprising that these standards are not suitable for Cronus control messages.

7.2 Design Issues

7.2.1 Objectives

The Cronus message structure conventions will be realized by a group of software components collectively called the Message Structure Facility (MSF). The Message Structure Library (MSL) is the realization of an MSF component, a library of functions or procedures which are available to processes on any Cronus host. Messages are composed by passing information to the MSL procedures; the result of a sequence of such calls is a data structure. This data structure can be transmitted from one process to another, and subsequently parsed by MSL procedures at the receiving process.

The objectives for the MSF relate to the MSL and the data representation; in approximate order of importance, they are:

1. **LOSSLESS STORAGE.** A process should be able to extract all of the information inserted into a message structure by the process which created it. The specification of the MSL interface should precisely define the data structures passed into and out of MSL procedures, so that this property has a clear and simple meaning to MSF clients.

2. EXPRESSIVE POWER. The message structure should capture enough of its clients' semantics (e.g., the size and type of data fields) for its use to be natural and convenient. Stated differently, clients of the MSF should not usually communicate and retain information relevant to message structure, except through the agency of the MSF.
3. PERFORMANCE. The data structures should be compact relative to the data they contain (e.g., less than 100% overhead for messages of 16 octets, and less than 25% overhead for messages of 100 octets or more). The MSL algorithms should be simple and have execution time linear in the message size.
4. PORTABILITY. The MSF concept should be portable to different language environments, e.g., the MSL could be coded easily in C, Pascal, or Ada. The C implementation in particular should be portable among all of the hosts in the Cronus ADM.

Attaining Objective (1) assures us that the MSF can be used to move an arbitrary data structure (viewed as a bit- or octet-vector) from one Cronus host to another. The representations of the data structures may differ at the sending and receiving hosts, but no information will be lost. For example, on the VAX a message may be stored as a consecutive sequence of 8-bit bytes, while on the C/70 the same message is stored as a sequence of 10-bit bytes.

Whether or not the MSF meets Objective (2) is unavoidably a subjective judgement. The potential uses of the MSF are diverse and unspecified. Because the MSF is accessible to application programmers, even a complete specification of the system requirements would not be sufficient to understand all of the implications of the design.

Objective (3) implies that efficient composition and parsing of very large messages (thousands of octets) is not a requirement. For small messages, it is acceptable for the MSL to locate fields by linear search and move fields by block transfers, operations which require time proportional to the message size. A structuring facility for large messages might well find these costs too high, and thus rule out the most straightforward implementation based on contiguous octet vectors. Further, if message fields were known to be large on the average (e.g., hundreds of octets for a paragraph of text or tens of thousands of octets for a bitmap graphics image), the message structure could use large field descriptors without increasing the percentage of storage devoted to overhead; this would open

many possibilities for more complex and efficient structure encodings. The small message assumption requires structural information to be small in absolute volume, for example, a message with 100 octets of client-supplied information should be smaller than 125 octets.

Portability, Objective (4), reduces the cost of the implementation on the eight or more hosts in the ADM. Large portions of the MSL will be portable among all of the ADM hosts supporting the C language. The host-dependent procedures must be recoded for each machine type (e.g., C/70, 68000, LSI-11) in the ADM, but the implementations will be very similar.

We did not make compatibility with any existing standard for message structure a goal because we know of no such standard that adequately complies with these four objectives.

7.2.2 A Taxonomy of Message Structure Conventions

A standard for message structure can be described as a point in a design space. Self-description, language integration, data type support, and performance are proposed in this section as useful axes in this space. These aspects and their significance to the MSF are described individually below.

A message is self-describing if it contains information about itself--about its own structure, or about the structure or type of its components. If we adopt such a convention, a receiver can depend upon the presence of this information, and need not rely upon higher-level protocols for its inclusion.

For example, a receiver might expect a message containing a timestamp; a timestamp might be represented either as a binary integer of 32 or 64 bits, or as a fixed length ASCII string. If messages contain no self-descriptive information, the receiver must make prior arrangement with the sender to either: 1) place exactly one of the possible formats (e.g., 32-bit binary) in every message; 2) indicate in each message which variety of timestamp was included. In case (2) the question of self-description recurs, over the representation of the indicator field.

The conventions for message structure considered in this note contain self-descriptive information which applies to the position, size, data type, and symbolic name of a message field, although not all of these are present in each convention.

Conventions for message structure can be influenced to a greater or lesser extent by the programming languages used to implement them. Tight integration might be achieved by developing a standard representation for a linguistic structure such as a Pascal record or C structure; weak integration is achieved by packages which strive for portability, and must be compiler or even language independent to a large degree.

Tight integration tends to improve performance, because the compiler's ability to optimize references to messages using structure-like objects defined in the language can be exploited. Such conventions may be convenient to use, because they blend well with features of the language; for example, expressions involving messages might be written in the language's standard syntax for structure accesses.

Tight integration has costs as well as benefits. It implies a strong dependence on a single language. Alternately, one must be willing to modify compilers to force message structures to conform to a standard representation. Certain concepts natural to message structures may be foreign to the data structuring facilities of the host language (e.g., inclusion of self-descriptive information), and these may be difficult to implement.

The weakest form of integration implies reliance on only a few language features, that are present in many languages. For example, a message structure library might use procedure calls as the only form of invocation, arrays of integers as the only data structure, and only stack-oriented storage allocation at procedure entry time. A library which obeys these constraints could be implemented in Pascal, C, PL/1, and most other block-structured languages. Furthermore, if portability is an important goal, the implementation for each language can be made portable across a range of compilers and host machines. Thus weak integration allows structure conventions to be implemented uniformly on many systems, at reasonable cost.

A library that is language-, compiler-, and host-independent imposes the burdens of integration on its users. For example, an application program may be forced to convert internal structures to message structures through a lengthy series of procedure calls, one field at a time. Programs will be more complex and execution slower if this approach is followed.

The conventions described in this note are language independent, in general. In order to mitigate the costs of the interface between message structures and data structures within a program, it is suggested that local data objects should sometimes be stored in the message format, i.e., the MSF can be viewed, to

some extent, as an alternative to the record or structure data types in the host language.

The degree to which the convention incorporates data type concepts is related to language integration. A language-based convention permits messages to contain some or all of the standard types defined in the language.

In the simplest case, a convention may consider messages to be composed only of bit- or byte-strings. The responsibility for interpreting the message fields as integers, character strings, etc., is left to higher-level software. A somewhat more complex convention may define the representations of basic data types (e.g., integers, booleans, and strings) in a language- or host-independent way. These data types may or may not include composite types (e.g., lists, records, arrays).

A convention may explicitly support the definition of new types, to be treated like the predefined types. There may be an administrative authority which guarantees the uniform interpretation of the types which evolve after the convention has been established.

If the application domain of the convention is well understood, data types especially important to that domain, may be included. Electronic mail systems are an important example; data types such as "phone number" and "network address" can be helpful here. Control traffic in a DOS might utilize a different set of conventional types, for example, Universal Identifiers, Transaction Identifiers, and timestamps in various formats.

Specification of data type representations is separable from the issue of self-description. A convention which specifies the representation of a 32-bit, two's complement integer, for example, may or may not include a type tag on elements of this type when they are embedded in a message structure.

Small and simple structures are easier to parse than large and complex ones. More complex conventions imply, in general, more complex and costly software packages to assemble and disassemble messages.

Execution time costs associated with message structures can be roughly divided into three categories:

1. The cost of transmission is an increasing function of message size.
2. The cost of composing and accessing the message is a

function of the complexity of the convention.

3. There is a cost borne by the clients for encoding higher-level concepts in those known to the convention.

If a convention is insufficiently rich in concept, (3) may be the dominant cost of use. If it is too complex, (2) may dominate. The most desirable situation is one in which (1) dominates, and furthermore most of the information content of messages is useful to the recipients.

The Cronus conventions selected have the following characteristics: 1) most operations on data structures are octet-oriented, and octet-oriented machine operations are efficient; 2) small data fields (e.g., an enumerated type with a few values) are represented very compactly, usually in one or two octets; 3) a key is stored with each data value to indicate its meaning or purpose--keys can be as short as one octet, or many octets (e.g., some keys may be symbolic names). The message structure routines are simplified because they contain no inherent knowledge of the keys.

7.2.3 Four Existing Conventions

7.2.3.1 NSWB8

The NSWB8 protocol [1] defines the structure of NSW control messages, thus its intended use is similar to that of the Cronus MSF. NSWB8 does not define a uniform client interface, but only the data structure.

An NSWB8 message is a string of octets; all fields of the message are octet-aligned. Each field is preceded by an octet designating the type of the field (i.e., one of empty, boolean, index, integer, bitstring, charstr, list, or pad); the length of a field can be inferred from the field's type. The primitive types empty, boolean, index, integer, and pad are fixed length, and occupy from 1 to 5 octets, including the type octet. The types bitstring and charstr contain a count (a 16 bit unsigned integer), followed by "count" bits (in ("count"+7)/8 octets) or "count" octets, respectively.

The type list permits NSWB8 messages to be recursively structured. A list begins with a type octet and a count, as for bitstring and charstr, but is followed by "count" fields of arbitrary type; some of these may again be of type list. NSWB8 is not closely tied to any programming language or host

architecture; the set of defined types is minimal, and does not include types (such as floating point) that are quite machine specific.

The most important deficiency of NSWB8 is the absence of a client interface specification to a standard library for manipulating NSWB8 structures. This interface is a necessary prerequisite for the construction of a portable MSL. Two further problems are 1) the amount of overhead for some fields (e.g., a variable length field has a 3-octet descriptor), and 2) the requirement that every receiver have knowledge of all defined types in order to parse an NSWB8 message because the size of a field is not explicitly coded in the field descriptor.

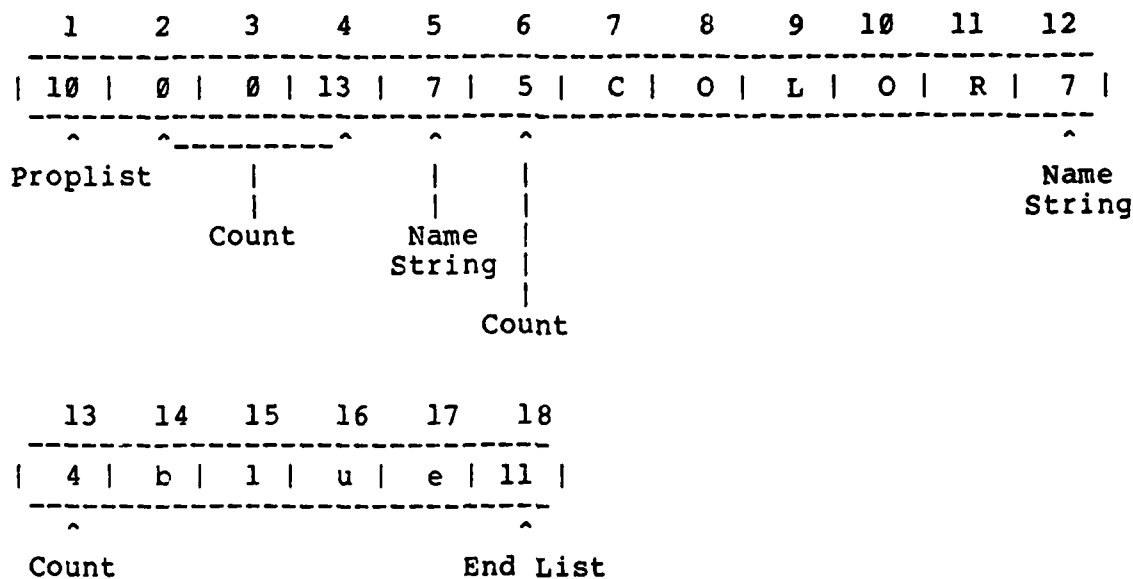
Finally, the NSWB8 protocol draws a sharp line between types defined in the protocol (e.g., integer, boolean, charstr) and new, client-defined types (e.g., an enumeration type); it does not suggest how the latter should be represented. In this respect the expressive power of NSWB8 is limited.

7.2.3.2 The Internet Message Protocol

The Internet Message Protocol (IMP) [2] is intended to be used for the transmission of electronic mail messages in the ARPA internet environment. The protocol includes conventions for multi-media messages as well as conventional ASCII text. Multi-media messages may contain digitized speech and video information; these fields tend to be extremely large (tens to hundreds of thousands of octets). In the ARPA internet, messages constructed are transmitted on TCP connections, and the transmission protocol imposes no limit on the size of messages.

IMP defines twelve basic data elements. A data element is a sequence of octets beginning with a type tag, which is an unsigned integer between 0 and 11. Nine of the data (No Operation, Padding, Boolean, Index, Integer, Extended Precision Integer, Bit String, Name String, and Text String) refer to atomic data elements without substructure visible to IMP. Three types (List, Proplist, End of List) are used to build composite structures. Composite structures may be recursive, e.g., a List may be an element of another List. The variable length elements (Padding, Extended Precision Integer, Bit String, Text String, List, and Proplist) contain a three-octet count immediately following the type octet. The Name String type has a one-octet count following the type octet, to reduce the overhead in the representation of short strings.

A simple example is the representation of the key-value pair COLOR:blue. Using IMP to represent this pair as the single element of a property list, the encoding would be:



A key always has type Name String, and the IMP defines a large set of well known key names significant in the electronic mail application domain, e.g., "NET", "OPERATION", "TYPE-OF-SERVICE", "DATE", "TRANSACTION".

Lists and property lists are powerful and convenient structuring concepts. The primary deficiency of the IMP, from the viewpoint of the Cronus MSF requirements, is the large overhead implied for small messages. An empty property list is five octets long; the smallest meaningful property list has a minimum of eight octets of structural information. The Cronus MSF is a descendant of the IMP, but is optimized for the compact representation of small structures.

7.2.3.3 The NBS Message Format

The National Bureau of Standards "Specification for Message Format for Computer Based Message Systems" [3], like the Internet Message Protocol, is intended as a standard for the format of electronic mail messages. These two standards have comparable domains of application, but two philosophical differences are apparent: 1) the Internet Message Protocol addresses the requirements of multi-media messages more directly; and 2) the NBS Message Format addresses broader issues, for example, the allocation of certain variable values for vendor-defined purposes.

A message conforming to the NBS Message Format is composed of data elements; there are 19 defined data types, 7 primitive types and 12 constructor that are used to combine elements and other data types:

Primitives

ASCII-String
Bit-String
Boolean
End-Of-Constructor
Integer
No-Op
Padding

Constructors

Compressed
Date
Encrypted
Extension
Field
Message
Property-List
Property
Sequence
Set
Unique-ID
Vendor-Defined

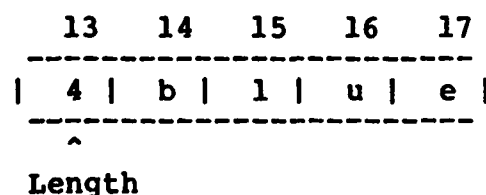
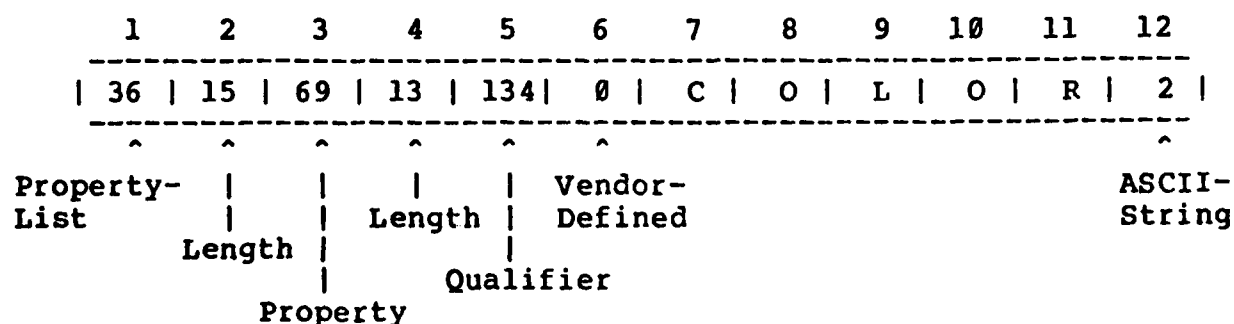
In its most general form, a data element is structured containing five fields:

1. Identifier Octet
2. Length Field
3. Qualifier Field
4. Property-List
4. Element Contents

The Identifier Octet specifies one of the defined data types, and

whether or not a particular instance has a Property-List (e.g., one instance of Integer may have a Property-List, while another does not). The Length field specifies the number of octets in the data element, after the Length Field. The Qualifier is present only for six data types (Bit-String, Field, Property, Compressed, Encrypted, Message), and is used to encode data-element-specific information, for example, the encryption algorithm for an Encrypted data element. The Property List binds properties to a data element; the only properties defined in the NBS Message Format are "Printing-Name" and "Comment". The Contents field contains the actual data represented by the data element.

A representation of the property "COLOR" with the value "blue" in a Property-List data element, assuming that COLOR is a vendor-defined property, is:



The amount of structural information (Identifier Octets, Length fields, Qualifier headers) in a message conforming to the NBS Message Format is comparable to that in a similar Internet Message Protocol message. One difference is that the NBS Message Format encodes many message field names as well known binary numbers (e.g., From=1, To=5, Reply-To=3), while the Internet Message Protocol represents the analogous field names as ASCII text. Thus messages in the NBS Message Format are potentially more compact, at the expense of greater static context in

programs that display messages to humans. From the viewpoint of Cronus control traffic, the difference is minimal, because the few field names are common to the domains of electronic mail and system control messages.

Like the Internet Message Protocol, the NBS Message Format is tailored to the electronic mail application, is oriented towards large messages, and defines only a class of representations. It does not define a client interface for manipulating data structures in the standard representation.

7.2.3.4 Courier

The Courier protocol [4], developed at Xerox Corp., is quite different from NSW8, the Internet Message Protocol, and the NBS Message Format. The Courier definition says that it "facilitates the construction of distributed systems by defining a single request/reply or transaction discipline for an open-ended set of higher-level application protocols." Layer Two of Courier defines the atomic and structured data objects or messages which can be built in accord with the standard.

Courier data objects carry very little self-descriptive information, instead relying upon global context for successful communication. The Layer Two of Courier defines 14 canonical external types, 7 predefined or atomic types, and 7 constructed types. Only the String and Sequence types include self-descriptive information. Sending and receiving processes must possess a common understanding of the message structure, including the type and length of each message field. For example, Courier encodes an Integer value in the range -32,768..32,767 in a 16-bit data object field, in two's complement representation. The encoding contains no explicit indication of type or length of the data object; a recipient must be aware that an Integer field begins at a particular 16-bit boundary within a message to extract in Network Unreachable formation from it. The 14 data object types specified in Courier are:

Predefined

Integer
Long integer
Boolean
Cardinal
Long cardinal
String
Unspecified

Constructed

Enumeration
Array
Sequence
Record
Choice
Procedure
Error

The origins of Courier data types are rooted in the concepts of programming languages rather than the context of electronic mail. The approach suggests a model of distributed programming in which the global context for message interpreting phase. A Courier message could be described by a type template which automatically drives the composition and parsing of messages. In fact, Courier specifies a grammar that could be used to define the class of templates, represented as ASCII character strings. The Courier standard talks about templates being created at "documentation time" for a software module. An example of a template for an enumeration type from the standard is:

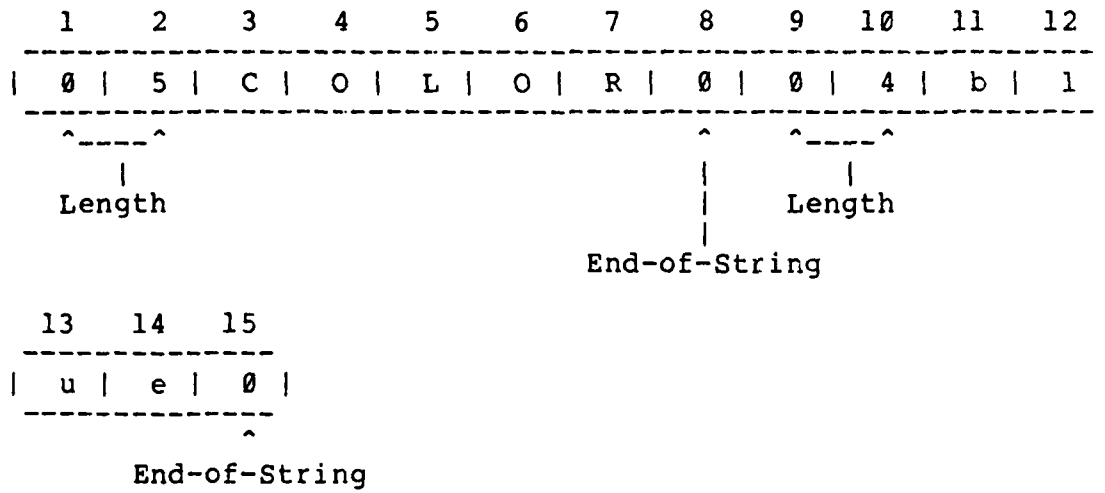
```
Mode: TYPE = {readPage(0),writePage(1),readAndOrWritePage(2)};
```

where a message of type Mode has three possible values, readPage, writePage, and readAndOrWritePage, which will be represented in a 16-bit field by the integers 0, 1, and 2, respectively.

Courier provides no direct encoding of key-value lists. The COLOR:blue example above could be encoded as a record composed of two String fields, using the Courier template grammar:

```
RECORD[Key,Value: String]
```

The corresponding representation of the pair COLOR:blue as a sequence of octets would be:



Courier messages can be extremely compact, since they contain little self-descriptive information, but proper interpretation of the messages depends upon pre-established context. This is true of any message but Courier would require more externally-supplied context in the domain of electronic mail applications, for example, than the Internet Message Protocol or the NBS Message Format. In general, a protocol which includes more self-descriptive information is a better choice for the Cronus project, because it reduces the mechanism supporting process-to-process communication.

8 Cronus System Libraries

The Process Support Library (PSL) is a collection of functions, that may be bound into the load image of a Cronus process. Only those routines actually needed by a process will be included in the load image. The data structures implemented by the PSL are within the address space of the process.

The PSL contains two important classes of function entry points. One class is widely used directly in application program development. The other class corresponds more or less to system calls in ordinary operating systems. These generally invoke a single operation on a particular type of object; the first class is generally implemented from components in this class.

During the period covered by this report, the general outline of the library has been prepared. Documentation of the principal object operation functions has been developed, and is currently being revised. Certain service routines, in particular, a portable i/o library, string, message structure, and data table manipulation functions have been written.

There are two packages available for data table initialization and maintenance. These insert, delete, and find key buffers and their associated value buffers. They are intended to be used in applications requiring many fast data retrievals.

The first of these, the LQH table maintenance package, creates an open address table, which is statically allocated by the calling program. It uses a linear quotient hashing scheme (i.e. double hashing if a collision occurs). These routines work fastest in a relatively unpopulated table.

The second, the BKH table maintenance package, gets storage as it needs it. It uses a bucket hashing algorithm, storing its values in linked lists. The linked list structure makes this package very effective for applications requiring many insertions or deletions.

9 Configuration Management

A configuration management plan for Cronus has been developed, and tools developed and acquired to automate aspects of this plan. The general principle of configuration management in the Cronus system is based on saving an audit trail of modifications to all source files (both program and English text), and of archiving obsolete program object files. The source file modifications, except those in the user manuals, are saved using the Revision Control System, which was obtained from Walter Tichy of Purdue. The manual revisions use the standard BBN-UNIX manual subsystem. Program objects are automatically saved by the `install` command.

The basic commands in the Revision Control system are as follows:

<code>ci</code>	checkin a file
<code>co</code>	checkout a file
<code>rcsident</code>	identify a version
<code>rcsdiff</code>	diff the current version with the RCS version
<code>rccs</code>	modify the rcs control and commentary
<code>rlog</code>	show selected portions of the file history

10 Standards, Policies, and Procedures

A Standards, Practices, and Policies Manual has been prepared for the Cronus Distributed Operating Project. Since the Cronus Advanced Development Model (ADM) is only the first instance of the Cronus DOS, the standards and practices described herein are designed to support the substitutability and portability goals of the project as well as to enhance the overall maintainability of the system.

The Cronus System is implemented on a heterogeneous collection of machines and constructed from a number of constituent operating systems.

11 System Documentation

11.1 User Manual

Documentation is an important aspect of Cronus development. We will be preparing a User Manual, an Operations Manual and Program Maintenance Manual describing the system from these various viewpoints. We have formulated initial plans for the development of these manuals.

The User Manual (UM) consists of those documents required to make effective use of the system. Many users will require only documents which describe the terminal interface to the system, others will need a description of the programming interface, while other users will require more detailed information. The complete UM is compiled from numerous documents from several sources. The following are components of the UM:

- o The basic user manual is a series of separate documents organized into sections which describe the manner in which commands may be invoked directly by a user, and the programming interface for standard library functions. This manual exists in online and hardcopy forms.
- o A Cronus Glossary of terms used in the various documents, particularly the User Manual, System/Subsystem Definition, Program Maintenance Manual, and Manual will be compiled.
- o There are reference manuals for the more complex commands and subsystems. These documents generally exist in hardcopy only, although the text generally exists in machine readable form as well.
- o User manuals for certain parts of the Constituent Operating systems may be included in the Cronus UM.

The online user manual will be maintained throughout the life of the project.

11.2 Operations Manual

The Operations Manual (OM) consists of a series of documents, including:

- o The Cronus Operations manual which describes procedures for operating the Cronus System, including startup, shutdown, crash recovery, and the interpretation of

console messages and other similar status and exception information.

- o The Operations manuals for each of the COSs.
- o The Operations manuals for other system components, for example, for the the internet gateway.

11.3 Program Maintenance Manual

The Program Maintenance Manual (PMM) consists of the information which is needed to understand and modify the various programs which comprise the Cronus DOS. Information contained in the Functional Description and in the System/Subsystem Description is essential to the comprehensive understanding of the system. can be considered part of the PMM. The Program Maintenance Manual is made up of a number of other documents, including:

- o The Cronus Standards, Procedures and Policies document defines the methods used in coding and documenting programs, establishes standard practices for the use of program libraries, and describes the configuration management used in the Cronus Project.
- o Manuals for the Constituent Operating Systems, specialized software packages used in the Cronus System, and for the hardware components in the Cronus systems. These are generally supplied by the vendors from whom the hardware or software was obtained.
- o Manuals, drawings, and other documentation as appropriate, for all hardware developed by BBN specifically for the Cronus project.
- o The Program Maintenance Tools are described in the PMM section devoted to the analysis of code.
- o The Program Code Analysis contains the results of applying the Program Maintenance Tools to the code, and also contains listings of the delivered version of the code, which provides unambiguous documentation of the system.
- o The System Notebook contains the series of informal documents, known as DOS Notes, accumulated during the life of the project.

REFERENCES

- [1] Postel, J., "NSW Data Representation (NSWB8)," USC/Information Sciences Institute, IEN 39, May 1978.
- [2] Postel, J., "Internet Message Protocol," USC/Information Sciences Institute, RFC 759, August 1980.
- [3] Deutsch, D., Resnick, Vittal, & Walker, "Specification for Message Format for Computer Based Message Systems," Bolt Beranek and Newman, Report No. 4486, August 1981.
- [4] "Courier, the Network System Remote Procedure Call Protocol," Integration Standard, Xerox Corporation, 1981.

INDEX

archive.....	39
audit trail.....	39
load image.....	38
Process Support Library.....	38
PSL.....	38
source.....	39
system call.....	38



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C³I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED

5-84

DTHC